KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science
Department of Electrical Engineering

Knowledge-driven development of telescope control systems

Wim Pessemier

Supervisor:
Prof. dr. ir. G. Deconinck
Co-supervisors:
Prof. dr. H. Van Winckel
Ing. P. Saey

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

March 2017

Knowledge-driven development of telescope control systems

Wim PESSEMIER

Examination committee:
Prof. dr. ir. C. Vandecasteele[1], chair
Prof. dr. ir. G. Deconinck[2], supervisor
Prof. dr. H. Van Winckel[3], co-supervisor
Ing. P. Saey[4], co-supervisor
Prof. dr. ir. H. Bruyninckx[5], assessor
Prof. dr. ir. J. Driesen[2], assessor
Prof. dr. ir. E. Motoasca[4]
Dr. ing. G. Raskin[3]
Dott. G. Chiozzi[6]

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

[1] KU Leuven, dept. of Chemical Engineering
[2] KU Leuven, dept. of Electrical Engineering (ESAT - ELECTA)
[3] KU Leuven, dept. of Physics and Astronomy, Institute of Astronomy
[4] KU Leuven, dept. of Electrical Engineering (ESAT - ETC, E&A), Technology Campus Gent
[5] KU Leuven, dept. of Mechanical Engineering (PMA)
[6] European Southern Observatory (ESO)

March 2017

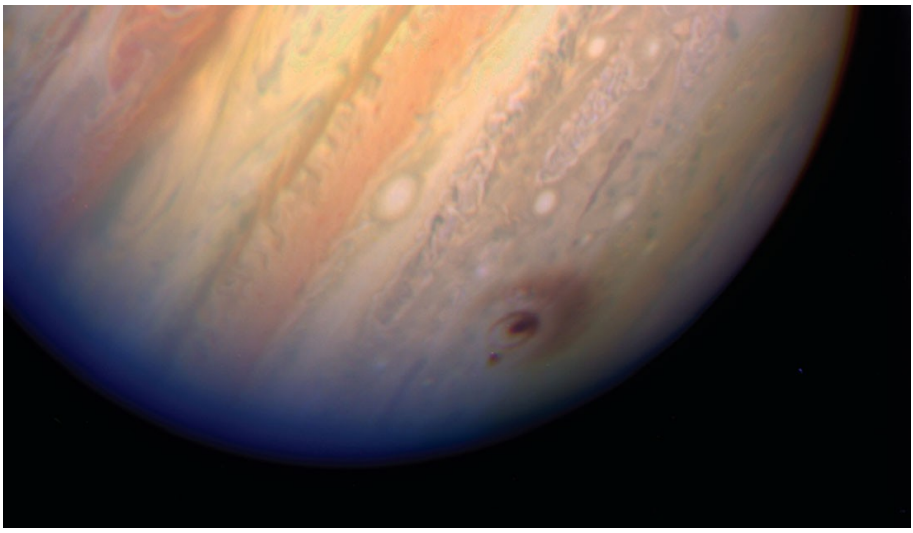
Cover photo: the Mercator Telescope at dawn.
Photo credit: Péter I. Pápics.

Uitgegeven in eigen beheer, Wim Pessemier, Celestijnenlaan 200 D, B-3001 Leuven (Belgium)
E-mail: wim.pessemier@outlook.com.

Preface

When comet Shoemaker-Levy 9 collided with Jupiter in 1994, two years after breaking apart due to tidal forces, I remember gazing at the impact through a telescope at Volkssterrenwacht MIRA. At that point, I knew that my pair of binoculars – which I had been using countless times to watch the Moon and the moons of Jupiter from my parent's backyard – did not satisfy my requirements anymore: I had to start saving my pocket money for a real telescope. Saving took a few years though, which fortunately were bridged by the bright comets of Hyakutake and Hale Bob, impressive sights to the naked eye in 1996-97. When I finally got my 15 cm reflector telescope at the age of 12, I could finally see the hundreds of individual stars in the globular cluster of Messier 13, which had remained a fuzzy patch on the sky through the binoculars. It would take me another few years before taping a webcam (with carefully modified electronics to allow long exposures) to the eyepiece holder of this telescope, revealing many more stars of the cluster – and even a trace of the distant galaxy of NGC 6207 in the background.

Galileo Galilei, the first astronomer that used a telescope to watch the planets and the stars, could never have witnessed an impact such as the one by Shoemaker-Levy 9. His telescope, equipped with a low quality lens of a few cm in diameter, wasn't even capable of showing the thick brown cloud bands on the Jupiter disk – which are easy targets for a small amateur telescope nowadays. Almost four centuries later however, numerous telescopes on Earth and in space were able to witness the cometary impact in great detail. Among them was the W. M. Keck telescope, one of the largest telescopes in the world, which used its primary mirror of 10 meters in diameter to collect the light of the violent shock waves, as the comet plunged into Jupiter's atmosphere. Despite the enormous progress in both science and technology, at that time there was no evidence that such planetary events could also occur outside our solar system. It wasn't until 1995, when Michel Mayor and his PhD student Didier Queloz of the Observatory of Geneva discovered *51 Pegasi b*, the first known extrasolar planet orbiting a sun-like star. At the time of writing this thesis, more than 3 400 exoplanets have been confirmed, some of which are earth-like planets in the habitable zone of their host star. In order to analyze the atmospheres of these planets, a new generation of "extremely large telescopes" is being built, thereby pushing the boundaries of science and technology ever further.



Impact of Shoemaker-Levy 9 (Hubble Space Telescope image, July 18, 1994).

Whether it is in the course of a person's life, or in the course of history, I'm convinced that most things advance by evolution, not revolution. As revolutionary as an extremely large telescope may seem, numerous of its technologies were tested before, and are the result of a long evolution of small steps. Similarly, as revolutionary as the catchy title of this thesis may seem, the full exploitation of a "knowledge-driven" development approach is still far ahead, and will undoubtedly require manifold small steps, of which I can only hope that this work is one. Certainly this work added a small step to the Mercator Telescope, in the form of a shiny new telescope control system, good to deliver a few extra photons each night to science, for many nights to come. Knowing that astronomers in the next years, from young PhD students to (well, older) professors, will experience the result of this work, feels satisfactory to say the least.

I wish to thank Geert, supervisor of this project, for guiding me towards this result with lots of patience, confidence, and good advise. Hans, you gave me the opportunity and the freedom to design and build a new control system for your pet telescope, not as fast as possible, but as good as possible. Thank you for allowing me to be part of the Mercator project, to get this PhD degree in the meanwhile, and to participate in the most interesting conferences (regardless of where they took place). Philippe, without your support I wouldn't have rolled into the Institute via my Master's thesis almost 10 years ago, and I wouldn't have ended up doing a PhD at ESAT. I'm also grateful to my assessors, Herman and Johan, for challenging me and for altering the course of this PhD project more than once. Finally also a big thank you to jury members Emilia, Gert, and Gianluca, for your relevant and constructive remarks. Gianluca, I'm very grateful that you wanted to participate in my project, and I'm very happy that we were able to talk and discuss things once in a different setting, in Leuven.

Of all persons at the Institute, of course my office mate (and jury member) Gert deserves a very big thank you. I don't know what it is exactly that makes us collaborate so well, perhaps it is because our interests are very complementary (you're more of a "hardware guy" and I'm more of a "software guy") and because at the same time we agree on so many things – about work but also about many other things. Jesus, thank you for all your help and friendship, and for the countless days that the two of us were working together at the observatory, I really enjoyed those days. I'm also grateful to Saskia, Florian, and Johan, for always helping out when needed, and for the nice atmosphere and company on La Palma or in Leuven. I also wish to thank all colleagues of the Institute of Astronomy, many of whom I had the pleasure to get to know more in person, while we were sharing the telescope on La Palma. The IvS has been a fantastic place to work during the past 9 years, I cannot recall a day that I did not enjoy.

Finally I also want to thank my parents, for always stimulating me to discover and learn and follow my dreams, and to always look ahead. The biggest thanks goes to Lotte, for always supporting me and for always being there for me – even if I was away again on one of the many trips for work, and even if I prolonged those trips again by traveling sometimes thousands of kilometers more "while I was there anyway". Together with Aaron (and our soon-to-be-born second little one), you always kept my feet on the ground, when my mind was a bit too far off in space.

Abstract

Since the very first observations of Jupiter's moons by Galileo Galilei more than four centuries ago, the size of ground-based optical telescopes has been growing steadily up to the current generation of telescopes with primary mirrors of 10 meters in diameter. The next generation of "extremely large telescopes" (ELTs) currently under preparation will almost quadruple that figure and will, for the first time, have atmospheric turbulence correction built-in by design. Inevitably, this leads to an increase in complexity and costs of the systems that control the behavior of the telescopes – the telescope control systems. Current practices therefore try to improve the reusability of the control systems design by developing reusable software frameworks, by applying model-based systems engineering, and by integrating off-the-shelf components. These efforts essentially foster the reuse of information within a single engineering discipline, across multiple engineering disciplines, and across different technologies, respectively.

Despite the ability of current practices to build even the largest telescopes in history, we identified three fundamental problems that are still present. Firstly, design information (from system-level requirements to detailed software models) is mostly captured in an informal way – even if popular modeling languages are used that promise the opposite – making it impossible for a machine or another person to interpret this information as it was originally intended. Secondly, the graphical representations currently in use add even more informality to this information. Finally, the prevalence of design patterns that were adopted from object-oriented programming restricts the reuse of information now and in the future. We call these problems "fundamental" because they are severely over-constraining the designs, making it very hard to apply solutions in the future, as the telescopes evolve through their projected lifetime of several decades. The goal of this thesis is to work out a solution that addresses those problems, in way that can be realized today. By specifying, implementing, and applying a fundamental methodology change to a real telescope – and not just a temporary test set-up in an "ideal" environment – we can analyze the feasibility and the readiness of such a solution.

The fundamental methodology change, promoted by this thesis, aims to shift the current practices from a traditional "model-driven" methodology towards a "knowledge-driven" methodology, in which *formal knowledge representation*

plays a pivotal role. In the framework that we propose, domain knowledge is captured by a set of ontologies: explicit and formal representations of the concepts and other entities in the domains of interest, and the relationships that hold between them. The ontologies that we created formally define the name and the meaning of a whole range of concepts needed to represent telescope control systems, from abstract finite state machines to the specific function blocks of the IEC 61131-3 industrial programming language. A software library called Ontoscript was developed to automatically map those concepts to the primitives of a set of internal domain-specific languages (DSLs) based on CoffeeScript. The telescope control systems can thus be "programmed" by a set of textual DSLs whose semantics are defined by a set of ontologies and by CoffeeScript. Finally, a tool called OntoManager was developed to offer a single interface to the modeling, reasoning, querying, and template-based artifact generation capabilities of the framework in a way that can be used by domain experts who are unfamiliar with formal knowledge representation.

OntoManager was used to develop a new control system for the Mercator Telescope, a 1.2 m optical telescope located at the island of La Palma. A total of 9 subsystems were modeled, from the "slow" pneumatic control of the primary mirror support to the much faster, time-critical and safety-critical motion control of the telescope's main axes. For each subsystem, a set of interconnected models represent the systems design (including requirements and systems breakdown), the electric design (from the device level down to the individual wires) and the software design (from the high-level function block declarations down to implementation of the interlock expressions). By reasoning and by querying these models, we were able to automatically generate system specification documents, verification reports and source code for the industrial control system in charge of the telescope and for the top-level control system in charge of the whole observatory. This has lead to the successful installation of the new Mercator telescope control system in June 2016.

Evaluation shows that the methodology change proposed by this thesis is feasible, for a real operational telescope of at least the size of the Mercator telescope, even within the constraints imposed by a PhD project. By synthesizing state-of-the-art practices in knowledge engineering, systems engineering, and software engineering we were able to address some fundamental problems of current practices in telescope control system design, thereby making design knowledge more reusable across the boundaries of engineering disciplines and technologies. Analysis of the real-world application with respect to the initial requirements of the framework reveals the added value of the methodology change, but also the limitations and pitfalls of the current implementation. We conclude this thesis by reasoning about the requirements and the possible benefits of generalizing our results to much larger telescopes, or even other application areas.

Samenvatting

Sinds de allereerste waarnemingen van Jupiter's manen door Galileo Galilei meer dan vier eeuwen geleden, zijn aardse telecopen alsmaar groter geworden, tot de huidige generatie van telescopen met een primaire spiegeldiameter van wel 10 meter. De volgende generatie "extremely large telescopes" (ELTs) die momenteel ontwikkeld wordt zal dit cijfer bijna verviervoudigen, en voor het eerst beschikken over een atmosferische turbulentiecorrectie die ingebouwd wordt van bij het ontwerp. Onvermijdelijk leidt deze evolutie tot een toename van complexiteit en kostprijs van de controlesystemen van de telescoop, die het gedrag van de telescoop moeten controleren. Het is daarom de gangbare praktijk om de herbruikbaarheid van de controlesystemen te verbeteren, door het ontwikkelen van herbruikbare software frameworks, door het toepassen van model-gebaseerde systeemkunde, en door de integratie van kant-en-klare ("off-the-shelf") onderdelen. In essentie wordt hierdoor de herbruikbaarheid bevorderd van informatie, over de verschillende ingenieursdisciplines en de verschillende technologieën heen.

Ondanks dat men er op deze manier in slaagt om de grootste telescopen in de geschiedenis te bouwen, identificeerden we drie fundamentele problemen van de huidige werkwijze. Ten eerste wordt ontwerpkennis (gaande van systeemvereisten tot gedetailleerde software modellen) grotendeels vastgelegd op een informele manier – zelfs al worden er populaire modelleringstalen gebruikt die het tegenovergestelde beweren. Dit maakt het onmogelijk voor een machine of een andere persoon om deze informatie te begrijpen op de manier zoals die oorspronkelijk bedoeld was. Ten tweede maakt de huidige grafische manier van voorstellen deze informatie nog méér informeel. Tot slot beperkt de grote hoeveelheid ontwerppatronen die overgenomen werden van object-georiënteerd programmeren, de herbruikbaarheid van de informatie, zowel nu als in de toekomst. We beschouwen deze problemen als "fundamenteel" omdat ze sterke beperkingen opleggen aan het ontwerp, waardoor het zeer moeilijk wordt om nog een oplossing toe te passen in de toekomst, wanneer de telescopen evolueren doorheen hun voorziene levensduur van meerdere decennia. Het doel van deze thesis is om een oplossing uit te werken die deze problemen aanpakt, op een manier die vandaag gerealiseerd kan worden. Door de ontwerpmethodologie te wijzigen, en deze wijziging te specifiëren, te implementeren en toe te passen op een échte telescoop – en dus niet op een tijdelijke testopstelling in een "ideale"

omgeving – kunnen we de uitvoerbaarheid en de inzetbaarheid van een dergelijke oplossing analyseren.

De fundamentele wijziging in de ontwerpmethodologie die deze thesis promoot, heeft als doel om de huidige manier van werken te verschuiven van een "model-gedreven" methodologie naar een "kennis-gedreven" methodologie, waarbij formele kennisrepresentatie een cruciale rol speelt. We ontwikkelden een raamwerk (een "framework") waarbij domeinkennis wordt voorgesteld door een verzameling ontologiëen: expliciete en formele beschrijvingen van de concepten en andere entiteiten in een relevant domein, en hun onderlinge relaties. De ontologiëen die we ontwierpen leggen de naam en de betekenis vast van een heel scala aan concepten die nodig zijn om de controlesystemen van een telescoop voor te stellen: van abstracte eindige toestandsmachines tot de specifieke functieblokken van de IEC 61131-3 industriële programmeertaal. We ontwikkelden een software bibliotheek genaamd Ontoscript, om deze concepten automatisch te koppelen aan de primitieven van een verzameling domein-specifieke talen gebaseerd op Coffeescript. De controlesystemen kunnen daardoor "geprogrammeerd" worden door een verzameling tekst-gebaseerde domein-specifieke talen en door CoffeeScript. Tot slot hebben we ook een toepassing genaamd OntoManager ontwikkeld, om domein experten – ook diegene die niet op de hoogte zijn van kennisrepresentatie – via één enkele interface toegang te bieden tot de mogelijkheden van het framework m.b.t. modellering, redenering, querying, en de generatie van documenten en broncode via sjablonen.

OntoManager werd gebruikt om een nieuw controlesysteem te ontwikkelen voor de Mercator Telescoop, een 1.2 m optische telescoop op het Canarische eiland La Palma. In totaal werden 9 subsystem gemodelleerd, van de "trage" pneumatische controle van de primaire spiegelondersteuning tot de veel snellere, tijdskritische en veiligheidskritische motion controle van de hoofdassen van de telescoop. Elk subsyteem wordt voorgesteld door een verzameling modellen over het systeemontwerp (inclusief de vereisten en opsplitsing van het systeem), het elektrische ontwerp (van de apparaten tot de bedrading), en het software ontwerp (van de declaratie van functieblokken tot de implementatie van zgn. "interlock" condities). Door de computer te laten redeneren en informatie op te vragen over de modellen konden we niet alleen automatisch documenten (zoals systeemspecificaties en verificatierapporten) genereren, maar ook broncode voor het industriële controlesysteem dat de telescoop bestuurt, en broncode voor het hogere-niveau controlesysteem dat het hele observatorium bestuurt. Dit leidde tot de succesvolle installatie van een nieuw controlesysteem voor de Mercator Telescoop, in juni 2016.

Evaluatie toont aan dat onze voorgestelde wijziging in de ontwerpmethodologie realiseerbaar is, voor een operationele telescoop die minstens zo groot is als de Mercator Telescoop, ondanks alle beperkingen die voortvloeien uit een doctoraatsproject. Door de laatste stand van zaken in knowledge engineering, systems engineering en software engineering te synthetiseren waren we in staat om enkele fundamentele problemen van de huidige ontwerpmethodologie van

de controlesystemen van telescopen aan te pakken. Daarbij verbeterden we de herbruikbaarheid van de kennis die nodig is om systemen te ontwerpen, over verschillende ingenieursdisciplines en technologieën heen. Aan de hand van de Mercator Telescoop konden we de resultaten van ons framework aftoetsen aan de initieel opgestelde vereisten, en konden we niet alleen de toegevoegde waarde van de methodologiewijziging aantonen, maar ook de voor- en nadelen van de huidige implementatie. We besluiten deze thesis door te reflecteren over de vereisten en de mogelijke voordelen van het framework indien het zou worden ingezet in grotere telescoopprojecten of in andere toepassingsgebieden.

Contents

List of symbols

$\top$	True
$\bot$	False
$\wedge$	Logical conjunction (and)
$\vee$	Logical disjunction (or)
$\neg$	Negation (not)
$\vdash$	Logical implication
$\rightarrow$	Material implication
$\leftrightarrow$	Material equivalence
$:=$	Valuation
$\mathcal{U}$	Until (STL operator)
$\Diamond$	Eventually (STL operator)
$\Box$	Always (STL operator)

List of abbreviations

AC	Alternating current
ACS	ALMA common software
ALMA	Atacama large millimeter/submillimeter array
AO	Adaptive optics
API	Application programming interface
ASCII	American Standard Code For Information Interchange
CAD	Computer-aided design
CAN	Controller area network
CASE	Computer-aided software engineering
COTS	Commercial off-the-shelf
CPS	Cyber-physical system
CPU	Central processing unit
CWA	Closed world assumption
DA	Dome access subsystem
DAML	DARPA Agent Markup Language
DB	Database
DC	Direct current
DDR	Double Data-Rate
DIKW	Data-information-knowledge-wisdom
DRE	Distributed real-time embedded
DSL	Domain-specific language
DVI	Digital visual interface
ELT	Extremely large telescope
EPICS	Experimental physics and industrial control system
ESO	European Southern Observatory
FOAF	Friend-of-a-friend
GMT	Giant Magellan Telescope
FTP	File transfer protocol
GPS	Global positioning system
GTC	Gran Telescopio de Canarias
HERMES	High efficiency and resolution Mercator echelle spectrograph
HMI	Human machine interface
HS	Hydraulics and safety subsytem
HTML	Hypertext markup language
HY	Hydraulics subsystem

ICALEPCS	International Conference on Accelerator and Large Experimental Physics Control Systems
IEC	International electrotechnical commission
IEEE	Institute of electrical and electronics engineers
IERS	International earth rotation and reference systems service
IP	Internet protocol
IRI	Internationalized resource identifier
JSON-LD	JavaScript object notation for linked data
KB	Knowledge base
KIF	Knowledge interchange format
KR	Knowledge representation
LIGO	Laser interferometer gravitational-wave observatory
MAIA	Mercator advanced imager for asteroseismology
MBSE	Model-based systems engineering
MEROPE	Mercator optical photometric engine
MOCS	Mercator observatory control system
MOF	Meta-object facility
MTCS	Mercator telescope control system
MTL	Metric temporal logic
MVC	Model view controller
NASA	National aeronautics and space administration
NC	Numerical control
NGC	New general catalogue
OCL	Object constraint language
OCS	Observatory control system
OIL	Ontology inference layer
OLE	Object linking and embedding
OMG	Object management group
OPC UA	OLE for Process Control – Unified Architecture
OWA	Open world assumption
OWL	Web ontology language
PC	Personal computer
PCI	Peripheral component interconnect
PG	Pumps group subsystem
PID	Proportional-integral-derivative
PLC	Programmable logic controller
POU	Program organization unit
PPS	Pulse per second
PTP	Precision time protocol
QUDT	Quantities, units, dimensions and data types
RAM	Random-access memory
RDF	Resource description framework
RDFS	RDF schema
RMS	Root mean square
SI	Système international d'unités
SKA	Square kilometer array
SLOC	Source lines of code

SPIN	SPARQL inferencing notation
SS1	Safe stop 1
SSI	Serial synchronous interface
STL	Signal temporal logic (see [62])
STO	Safe torque off
SysML	Systems modeling language
TA	Telescope axes subsystem
TAI	International atomic time
TC	Telescope cover subsystem
TCP	Transmission control protocol
TCS	Telescope control system
TE	Telescope encoders subsystem
TI	Time service subsystem
TMT	Thirty meter telescope
TT	Telescope telemetry subsystem
TTL	Transistor-transistor logic
UAF	Unified architecture framework
UDP	User datagram protocol
UI	User interface
UML	Unified modeling language
UNA	Unique name assumption
UPS	Uninterruptible power supply
URI	Uniform resource identifier
URL	Uniform resource locator
UTC	Coordinated universal time
VLT	Very large telescope
VNC	Virtual network computing
XMI	XML metadata interchange
XML	Extensible markup language
XSD	XML schema definition

Chapter 1

Introduction

OBSERVATIONAL astronomy is the branch of astronomy that uses observations of celestial objects to gain understanding of the universe around us. From the discovery of Jupiter's moons by Galileo Galilei in 1609, to the observation of gravitational waves by the LIGO and Virgo experiments in recent years, it has emerged as a scientific subfield that is heavily supported by – and often the driving force behind – advances in engineering to set up ever larger and more complex observatories. The synergy between the very latest scientific knowledge and engineering knowledge has led to large modern ground-based observatories, operating from the radio to the infrared wavelengths, as single telescopes or as large arrays of jointly operating telescopes.

1.1 Telescope control systems

The behavior of these telescopes is controlled by a *control system*, the focus of this thesis. A control system can be defined very generally as "an interconnection of components forming a system configuration that will provide a desired system response" [23]. The boundary between the control system and the physical system consists of sensors (such as temperature sensors and position encoders) and actuators (such as electric motors and pneumatic valves). Sensors and actuators are generally not considered as part of the control system, even though they do affect the behavior or response of the system. For instance, the maximum velocity of a telescope may be affected by the number of pole pairs of its motors. In the context of telescope control systems, we therefore adopt the more practical interpretation of the term "control system" by ICALEPCS, the leading biennial International Conference on Accelerators and Large Experimental Physics Control Systems, stating that it includes [47]:

- all components or functions, such as processors, interfaces, field-busses, networks, human interfaces, system and application software, algorithms,

architectures, databases, etc.

- all aspects of these components, including engineering, execution methodologies, project management, costs, etc.

In a broader sense, telescope control systems are examples of *distributed real-time embedded (DRE) systems*, used in the particular application area of observational astronomy. The term *embedded* here reflects the tight interaction between the software and the physical environment of the system via sensors and actuators, while *real-time* implies that time constraints must be met to satisfy the required behavior. For instance, to track a celestial object, a telescope control system receives feedback from its encoders, it calculates the trajectory of the object and the position control algorithm output, and it sends this output to the telescope drive system, all within a strict time frame. Finally the term *distributed* applies to the execution environment of the embedded system, consisting of several nodes that are in locally or geographically separated locations [35]. Control systems of large telescope arrays such as ALMA (the Atacama Large Millimeter Array) may be distributed over an area of several kilometers in diameter.

A modern vision on DRE systems focuses even more on the tight integration of computational and physical elements, resulting in so-called *cyber-physical systems (CPS)*. Cyber-physical systems could be considered as a general term, not bound to a particular technology (such as the Internet-of-Things or IoT) or a particular application area (such as Industrie 4.0). They are defined as "integrations of computation and physical processes", consisting of "embedded computers and networks to monitor and control the physical processes" [58]. A tight integration of computation and physical processes requires understanding of the *joint* dynamics of computers, software, networks, and physical processes [60]. As these physical processes are compositions of many parallel processes, one of the key challenges of CPS design is the decentralized coordination of these concurrent processes. Addressing this challenge falls outside the scope of this thesis, as even the future largest telescopes such as the Thirty Meter Telescope (TMT) and the European Extremely Large Telescope (E-ELT) are envisioned to be centrally coordinated, having a (mostly) hierarchical control flow [104] [52]. Vital to the decentralized coordination between systems in a CPS, however, is the availability of *knowledge* about the systems. Embedded systems should not only exchange meaningless (contextless) data and information, but also information *about* this information. If such "meta-information" is sufficiently available and commonly understood, then future "intelligent" systems can interpret the exchanged information, reason about it, and act upon it. As will be seen in the next chapter, the methodology described by this thesis attempts to make some of this contextual information explicit. Even though – in its current implementation – we only use this contextual information at design-time (e.g. to increase reusability and consistency of the design) and not at run-time, it represents a modest step towards fulfilling one of the requirements of future cyber-physical systems.

1.2 Evolution

As the frontiers of astronomy are continuously evolving, so is the size of the telescopes that push these frontiers further. Figure 1.1 illustrates how the size of ground-based optical telescopes have grown since Galilei's first discoveries more than four centuries ago. This evolution is set to accelerate even slightly, as three future telescope projects will advance astronomy from the "very large telescope" (VLT) era to the "extremely large telescope" (ELT) era, with primary mirrors of over 20 m in diameter.

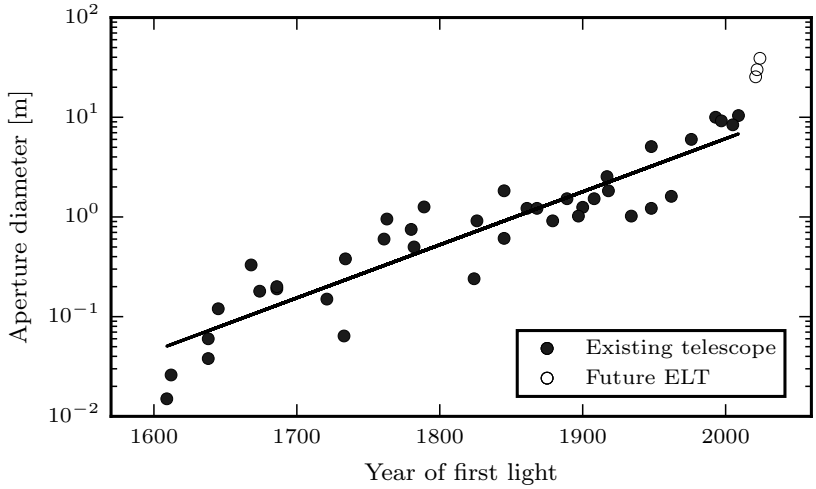


Figure 1.1: Aperture of ground-based optical telescopes in the last 4 centuries. (Data taken from Wikipedia's "list of largest optical telescopes historically – by historical significance" [122])

1.2.1 Complexity

At this point, a distinction should made between size (indicated by the number of elements of the system) and complexity (indicated by the number of relations between the elements of the system). Three cases can be discerned for the complexity of telescope control systems, as the telescopes grow in size:

1. The complexity of some parts of the system remains unaffected. Larger physical systems do not necessarily introduce more components to control, and hence do not necessarily affect complexity. For instance, the motion control system of ELTs are very similar to those of VLTs or even smaller telescopes, as in each case there is obviously still only one azimuth and one altitude axis. Even the pointing precision requirements of these axes remain unaffected, since sub-arcsecond pointing corrections are handled by additional optical elements instead of the mount itself. It means that

the complexity of controlling the axes of large telescopes is very similar to the complexity of controlling the axes of small telescopes.

2. The complexity of some parts of the system increases linearly. Larger telescope mirrors, more focal stations, larger arrays of telescopes, etc. can introduce a big increase in number of components to be controlled, but complexity most often increases in a very similar way. For instance, the primary mirror of the E-ELT will consist of 798 hexagonal segments (with almost 2500 actuators and 5000 pairs of edge sensors) [25], whereas the current largest segmented mirror in the world (of the Gran Telescopio de Canarias, GTC) only consists of 36 segments. Still, the complexity of the position control system of the mirror segments roughly grows linearly with the number of segments, since the position of each segment is only affected by their direct neighbors (via the edge sensors) and not by all other segments.

3. The complexity of some parts of the system increases more than linearly. Adaptive optics (AO) is one of the rare examples where a linear increase in number of components (e.g. the number of wavefront sensors and deformable mirrors) introduces non-linear effects on the complexity of the control system. AO also disrupts the typical hierarchical "top-down" control flow of a telescope, since a wavefront measured at a low level (e.g. by the instrument) may cause corrective actions to be taken by deformable or fast tip-tilt mirrors at other levels (e.g. by the telescope).

More in general, it can be said that as telescopes and their control systems continue to grow in size (sometimes by a huge step such as in the case of ELTs), their complexity – in most cases – follows a similar trend. Only rarely does complexity not scale with size, such as in the notable example of adaptive optics for future ELTs [16].

1.2.2 Costs

The continued increase in size and complexity of telescopes inevitably leads to increased development and maintenance costs. For example, in the VLT era, it was estimated that the total cost of a telescope scales with the diameter (D) of the primary mirror to the power of 2.6 according to Bely [3] or 2.7 according to others [101]. These figures result from the combination of two effects: some parts of the telescope such as mirror area scale with $\sim D^2$, while for example enclosure mass and cost rather relate to the area ($\sim D^2$) and volume ($\sim D^3$) of the enclosure (which are functions of the telescope tube length and hence also the mirror diameter) [3]. Bely further argues that the introduction of new technologies does not affect the exponent of the power law, but rather shifts its normalization. For instance, the introduction of alt-azimuth telescopes resulted in a cost reduction compared to equatorially mounted telescopes, but the cost of alt-azimuth telescopes still scales with $\sim D^{2.6}$ [3]. More recently however, more accurate cost models have been developed, based on a large

number of variables and their estimated scaling factors. For the future Thirty Meter Telescope (TMT) it has been estimated that only a very small fraction of components scales with a factor larger of 2 or greater, and that the overall cost may roughly scale to the mirror diameter with an exponent as low as 1.2 (when not extrapolating the mirror diameter too much) [78].

Budget allocations of large telescopes reveal that control systems represent only a small fraction of this cost. For instance, for the future E-ELT it is estimated that the share of the control system accounts to less than 5% (or € 47 M) of the total construction cost [25]. A recent cost comparison by Guzman based on three large existing telescopes yields a similar figure (4 to 5%), but it only takes the costs of the software development into account [40]. Guzman also estimates that software maintenance represents a significant cost over the lifetime of the telescope, accounting to roughly two thirds of the total life cycle costs of the software. Given a lifetime of 20 years and a ratio of maintenance to development costs of 2:1, it implies a yearly maintenance budget of 10% of the initial development budget.

Regardless of their exact development and maintenance cost functions, it is clear that – given their growth in terms of size and complexity – development costs and (more substantially!) maintenance costs of telescope control systems remain very important design variables besides the technical and scientific requirements.

1.2.3 Qualities

Increasing development and maintenance costs force engineers to use resources (such as materials, labor, time, energy, knowledge, ...) more efficiently. For instance, the 798 segments of the E-ELT primary mirror force engineers to create segment designs that are highly scalable and reusable, easy to access, to test and to maintain, robust to certain sensor and actuator failures, contributing to an overal telescope availability of at least 97% [111]. Increasing size, complexity and costs imply that these "qualities" such as scalability, reusability, accessibility, testability, maintainability, robustness, availability, etc. become increasingly important. They are sometimes referred to as the *non-functional requirements* (or even the *"-ilities"*) of a system, but as noted by Glinz [33], this terminology is vague and is used inconsistently in literature. According to Gliz, a quality discussed in this section is a *specific quality requirement*, or a requirement "that pertains to a quality concern other than the quality of meeting the functional requirements".

Reusability

An obvious way to use resources more efficiently, is to reuse them. Reusability (the ability to reuse existing assets) mitigates the scaling effects of costs and complexity of telescope control systems, as they grow in size. It is a quality already found everywhere in current telescope control systems, from the

systematic application of standards (e.g. ethernet for communication) to the development of general purpose artifacts that are reused for multiple specific purposes (e.g. a logger software library that is commonly used by the various subsystems of the telescope). Two more examples of reuse that are challenging, and of particular interest for this thesis and for embedded systems in general, are explained below.

Firstly, an inherent problem of embedded systems is the need to **integrate information across different engineering disciplines**. For example, consider the gear transmission between an electric motor and the telescope azimuth axis. This transmission ratio (e.g. 1 : 1440 for the Mercator Telescope) not only affects the mechanical design, but also the general system performance (e.g. for verifying the maximum velocity of the telescope), the electrical design (e.g. for selecting a suitable motor) and the software design (e.g. for generating the motor position setpoints). As will be explained in the next section, current practices put forward "model-based systems engineering" as the best way to make such information reusable across multiple engineering disciplines, by including the information in a common systems model.

Secondly, there is a need to **integrate information across different technologies**. Large embedded systems depend on a mix of technologies, combining custom and off-the-shelf hardware and software. The reuse of information across the boundaries of these technologies is challenging, and – as will be seen in the next sections – is hampered by current practices of telescope control system development. Nevertheless, it is important because if more information can be reused, then the system becomes less dependent on particular technologies. The systems will therefore become more able to evolve because technologies can more easily be replaced (e.g. due to obsolescence or changing requirements) and the overall homogeneity will increase.

Consistency

Reuse of information leads to more consistency. Designs that are consistent do not contain contradictions: for instance, the transmission ratio between the azimuth motor and telescope axis must be the same for all (mechanical, electrical, software, ...) designs. If the ratio is changed in one design, this change must also be reflected in the other designs. An obvious way to accomplish this is by defining the transmission ratio only once in one design, and by allowing all other designs to refer to it. When reusing information (e.g. across different engineering disciplines or across different technologies), the meaning of the information becomes very important, in order to avoid inconsistencies. For instance, if the concept of the "azimuth axis" is commonly used by multiple engineers, it must be very clear what an "axis" really stands for.

Traceability

Traceability in embedded systems design mostly refers to *requirements traceability*, which has been defined by Gotel and Finkelstein [36] as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)." Complex (telescope) projects may document thousands of requirements, which must be satisfied by the designs created by large international consortia. The ability to trace a requirement (such as the azimuth axis minimal velocity during pointing) to the design elements (such as the maximum motor velocity and the transmission ratio), and the other way around, is important to keep the design consistent and evolvable. Consistent, because it becomes easier to see when a design element is in conflict with a requirement. And evolvable, because it becomes easier to replace a design element if it can be traced back to the requirements that will be affected. It means that all information (about both the requirements and the design elements) must be reusable.

Verifiability

The ability the verify the consistency of a design is called the verifiability. For designs to be verifiable, the information they provide must naturally be reusable. Due to the difficulty of reusing information across the boundaries of engineering disciplines or technologies, verification is often confined to the designs of a particular engineering discipline or a particular technology. Cross-domain verification in embedded (and thus multidisciplinary and often heterogeneous) systems is much more challenging [12], but is becoming increasingly important as the systems grow in complexity.

Evolvability

Much related to the previous qualities, evolvability is the ability to respond effectively to change [108]. For a system to be evolvable, its elements must be easily traced back to their requirements, and its consistency must be verified easily. Because both scientific drivers and technologies advance significantly during the development time and the lifetime of a telescope (more than four decades all together in the case of ELTs!), modern telescope control systems have to be designed in a way that they can be adapted or extended without too much effort in the future.

1.3 Current practices

To be able to deal with increasing size, complexity, costs and qualities, the design methodologies of telescope control systems have evolved significantly over the years. Based on the available literature[1], three trends are clearly visible: the increasing importance of reusable software frameworks, of model-based systems engineering, and of off-the-shelf (third party) components.

1.3.1 Software frameworks

Software frameworks such as EPICS[2] and ACS[3] are used by dozens of existing large telescopes, particle accelerators and other large experimental physics facilities around the world. They are so-called "technical" frameworks because they effectively isolate a number of *technical concerns* (services) that are application-independent (such as logging, configuration management, alarms and events, etc.) from the *functional concerns* that are application-specific. Currently existing frameworks force the control systems into a so-called "three-tier" hierarchical architecture of high-level coordination systems at the top tier; low-level real-time control units at the middle tier; and devices with limited degree of intelligence directly connected to hardware at the bottom tier [16]. Each layer may consist of components, typically with few dependencies between each other, explaining the hierarchical control flow. Future ELTs are now challenging the reusability of these existing frameworks however, because of several reasons (described in more detail by Chiozzi et al. [16]):

1. the large scale of ELTs imply that the control systems must be able to monitor and control many more inputs and outputs (e.g. the E-ELT is estimated to contain over 100 000 I/O points, a 10-fold increase compared to current 8-10 m class telescopes);

2. the increased complexity of ELTs due to adaptive optics imply that the control systems must allow much more strict real-time coordination between processes of different (physically separated) subsystems;

3. the long expected lifetime of ELTs imply that the control systems must be highly adaptable to future technologies.

[1] Paper publications about ground-based telescope control systems are roughly confined to the proceedings of two alternating biennial conferences: *SPIE Astronomical Telescopes and Instrumentation* (with conference tracks dedicated to 'Software and Cyberinfrastructure for Astronomy' and to 'Modeling, Systems Engineering, and Project Management for Astronomy') and *ICALEPCS*, the International Conference on Accelerators and Large Experimental Physics Control Systems. *Experimental Astronomy* and more recently the *Journal of Astronomical Telescopes, Instruments and Systems* contain some additional papers, but it appears that the SPIE and ICALEPCS proceedings have by far the widest coverage.

[2] Experimental Physics and Industrial Control System [19]

[3] ALMA Common Software [15]

While the software architectures of the future ELT control systems are still in their conceptual phase, it appears that they will represent natural evolutions (rather than revolutions) of the currently existing frameworks (see [105], [99], [28] and [34]). Multi-tiered architectures with hierarchical control flows are still the norm, but additional coordination between subsystems (e.g. between the AO system, the instrument and the telescope) imposes tighter constraints on the infrastructure. This coordination may be based on state analysis in case of the E-ELT [52], or actor-based frameworks such as Akka in case of the TMT [104]. The Giant Magellan Telescope (GMT) project appears to deviate from this traditional hierarchical control scheme, by describing a process as a two dimensional grid of "steps" (a sequence of well defined states in time) and "tracks" (concurrently running autonomous components) via a "Workflow" specification DSL (Domain Specific Language) [28].

All three ELT projects further explicitly mention that the frameworks need to provide a means to *model* various functional aspects of the control system, in order to make the knowledge about these aspects more reusable and manageable. This will be elaborated in the next section about model-based systems engineering. Lastly, all three ELTs recognize the need to abstract technologies more rigorously compared to the traditional software frameworks (e.g. the frameworks must be "middleware-neutral"), in order to migrate more easily from one technology to the next over the lifetime of the project.

1.3.2 Model-based systems engineering

While software frameworks represent very explicit ways to reuse existing assets, much of the knowledge needed to build and maintain the control systems is still "hidden" in the software code, or is described in text documents, spreadsheets, wiki's, requirement databases (such as DOORS[4]), etc. Knowledge captured in these formats is very little reusable. Modern telescope projects therefore attempt to *model* some of this knowledge, so that it can be reused for various purposes. For instance, the state charts explored by ESO [52] may be used for verification purposes, but also to generate software code, to generate systems documentation, and to trace the system states to the system requirements (modeled by requirement diagrams). The reusable nature of models is especially valuable in systems engineering, because there the knowledge of several domains must be integrated. Model-based systems engineering (MBSE) is the general term used to describe the methodology that promotes models as the primary means to represent and to exchange information for systems engineering purposes. In practice we see that MBSE is often associated with SysML, the Systems Modeling Language, considered by some authors to be the "key enabling" technology for MBSE [50]. Nearly all future major telescope projects (including the E-ELT, TMT, GMT and SKA) have adopted SysML for modeling their systems requirements, their structure and/or behavior [49]. Of those projects,

[4]Dynamic Object Oriented Requirements System, a popular commercial requirements management tool by the company Rational.

the GMT appears to use SysML only complementary to a set of in-house developed DSLs with a textual notation [28].

1.3.3 Off-the-shelf components

Control systems for large experimental physics facilities (such as telescopes, particle accelerators, fusion reactors, ...) have, historically, been slow adopters of commercial off-the-shelf (COTS) industrial components and standards. Increased complexity and the associated development and maintenance costs have changed this picture: for instance, Myers and Salter of the worlds largest particle physics laboratory CERN concluded in 2005 that the difference between control in experiment systems and most industrial systems, has become mainly one of size (data acquisition excluded) [71]. At the same time, facilities are not only looking at industrial solutions, but also at generally available open-source software. To minimize the dependence on specific technologies (and the risk of vendor or technology lock-in), two approaches are being pursued: i) the use of cross-vendor standards and ii) the encapsulation of the third-party components. International standards such IEC 61131-3[5], OPC UA[6], and EtherCAT[7] are supported by multiple vendors and third-parties (hence reducing the risk of vendor lock-in) and often can be integrated more easily into the project's software frameworks because their specifications are publicly available. But as mentioned in 1.3.1, even these standards evolve or disappear over time. As a result, software frameworks often encapsulate the third-party components by providing a mapping (e.g. as a wrapper library or a proxy server) between the technology and rest of the framework.

1.4 Current problems

The current practices described above are the result of a long evolution (not revolution). They represent the "best practices", incrementally developed and refined by large specialized organizations, based on decades worth of experience. A commonality between these practices is that they are mature, well tested, and technically capable of controlling even the largest telescopes ever built by mankind. Nevertheless, a number of problems can be identified, which are affecting the current practices. Those discussed in this section are very "general" problems related to the design of complex embedded systems. Thus, they are not specific to telescope control systems, but they are shared by engineers in all kinds of (industrial) application areas. So while the previously described current

[5]IEC 61131-3 is an international standard defining PLC (Programmable Logic Controller) programming languages.

[6]OPC UA (IEC 62541) is the 'Unified Architecture' (UA) version of the original OPC (Object Linking and Embedding for Process Control) communication standard.

[7]EtherCAT (standardized in IEC 61158) stands for Ethernet for Control Automation Technology, a communication protocol invented by Beckhoff Automation but nowadays used by a large number of vendors.

practices are capable of meeting even the most stringent technical requirements of ELTs, they are still only partially addressing some of the inherent problems of complex embedded systems design. More fundamental methodology changes may be more appropriate (or even required) in the future, to address the three problems outlined in the remainder of this section.

1.4.1 Informal knowledge representation

All future ELTs are partially designed using SysML, which is an extension of a subset of UML 2, the second version of the Unified Modeling Language. Thus, SysML reuses the semantics and notations of a part of UML, and defines new semantics and notations specific to the field of systems engineering – again using UML constructs. It means that SysML is heavily rooted in UML, borrowing not only its good parts, but also its bad parts.

These bad parts (or the "ugly" ones, depending on which expert is talking [43]) have been subject to an extensive amount of research since the original publication of UML in 1997. One of the main criticisms of UML is its lack of semantics, and therefore "multiple and potentially contradictory interpretations of one and the same model are not excluded, and automatic interpretation must be hard coded in some way or another in the tool chain" [11].

A multitude of examples of UML constructs with deficient semantics are described in the literature (e.g. see [96], [22], [11], [98], [97]). A notable example is the confusion about "composite aggregation" (the black diamond symbol), sometimes called composition, ownership, containment, by-value, whole-part or strong aggregation [43]. In [44], Henderson-Sellers and Barbier list a total of 5 primary characteristics, 10 secondary characteristics and 5 consequent properties to describe "aggregation" (both by-value and by-reference). These characteristics include, for instance, irreflexivity ("a given object cannot be both whole and part at the same time"), shareability ("the ability of the part to belong to two or more wholes at the same time"), separability ("piece(s) can be removed from the whole without destroying either"), etc. The UML specification does specify some of these characteristics for the "composite aggregation" relationship informally, but leaves many others up to the interpretation of the user. According to Cook [43], it means that since UML by itself does not provide precise meaning, it must be added externally in a manner specific to the context at hand. For instance, a composition relationship in a *class diagram* may have a different meaning in the context of C++ programming (in which it would be interpreted as containment by-value, with parts existing only during the lifetime of the whole) compared to the context of Python programming (in which "everything is a reference" and the lifetimes of the parts are determined by reference counting [102]). Cook further notes that the lack of an exact meaning is not necessarily a disadvantage, if the language would be flexible enough to tailor it to a particular domain. Applied to the C++/Python example, if UML would *less* constrain the meaning of composite aggregation, then at least it could be *reused* by C++-specific and Python-specific extensions without violating the specification. Cook therefore

concludes that UML is trying to have it both ways but fails to succeed in either: its semantics are too imprecise to be usable for interchanging meaningful models, but at the same time they are too constrained to be usable as a foundation for domain-specific extensions. The result is that modelers typically ignore any UML semantics and invent their own [11]. Any additions (e.g. by stereotypes or tagged values) will suffer from the same problem, since UML does not offer a mechanism to precisely describe the meaning of these additions within the language itself [41]. Thus, the exact meaning of a class diagram in a C++ or Python context is in practice not formalized, but it is simply implemented in a tool-specific way (e.g. by a code generator).

Because its semantics are mostly formulated in natural language (aside from some "well-formedness" (i.e. syntactic) rules written in OCL, the Object Constraint Language), it is simply impossible to verify the consistency of a given model against the UML specifications. Logically, SysML is very similar in this respect, as it is mentioned explicitly in the latest SysML specifications (version 1.4) [77]:

> SysML is specified using a combination of UML modeling techniques and precise natural language to balance rigor and understandability. Use of more formal constraints and semantics may be applied in future versions to further increase the precision of the language.

SysML thus extends UML (affirming its semantic deficiencies) by introducing new primitives which are also defined by natural language and some "syntactic" OCL constraints. As an example, we quote the definition of a Requirement by the same specifications:

> A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition that a system must satisfy. Requirements are used to establish a contract between the customer (or other stakeholder) and those responsible for designing and implementing the system.

Several questions can be raised when trying to interpret the above definition. What's the difference between a "capability", a "condition", a "function" and a "performance condition"? How do they relate to a system? How does a "requirement" relate to a "contract" (or in other words, what kind of relation does "are used to establish" represent)?

These questions illustrate that "precise natural language" (as quoted from the specifications) is very hard to achieve in practice. Informal descriptions lead to ambiguous interpretations by humans, and even more so by machines. Reasoning about such a model or verifying its correctness is therefore bound to be tool-dependent. It means that the reuse of information across the boundaries

of engineering disciplines or across the boundaries of technologies, may lead to inconsistent models because the information may be interpreted differently on each side of the boundary.

1.4.2 Graphical notation

Other problems described in literature are related to the graphical notation of SysML and UML. One of the "good parts" of UML is that is a standard, or as noted by Mellor in [43]: a both technical and political endeavor, implying compromise and negotiation. It was the original intent of UML that made it so popular: to provide a *common* notation system for software development [103], one that can be reused by multiple persons to convey some information. However, the very same notation can be problematic when used outside of this original scope. Because, as noted by Fowler [30], how *standard* is this standard notation exactly? If the semantics of the UML and SysML primitives are not precisely defined (see 1.4.1), how can the graphical symbols that are "mapped" to these primitives then accurately convey information? Naturally they cannot, and therefore it relegates UML to being a mere sketching language according to some authors [43]. It works in both directions: not only can a concrete model be interpreted in various ways, but also a concrete idea can be modeled in various ways. While this observation holds true for both textual and graphical notations, the latter is more problematic due to the reasons outlined below.

A first issue concerns the capability of a graphical language to represent knowledge. How much information can be conveyed by squares, lines, pyramids, balloons, etc.? Looking at a typical SysML model[8], it is clear that much information is actually represented as text, either as informal comments or as formal (OCL) constraints. Furthermore, as elaborated in [24], the graphical notations that are used often lack "semantic transparency": they are too distant from the semantic concepts that they represent, and are therefore difficult to understand.

Additionally there is the danger that graphical models convey more information than what is intended by the author. For example, the size, color and location of symbols have no formal meaning in UML, yet they can (intentionally or not) encode information very effectively[9] [70]. For instance, it is very natural to draw a hierarchical structure in a vertical lay-out, with each level separated by equal space and perhaps highlighted by a specific color, and with the lower-level elements drawn smaller than the higher-level elements. The "implicit" semantics of such a visual representation are very powerful when communicating information between humans – they are the very reason that makes visual modeling so attractive. However, the same semantics are much more difficult to be interpreted by machines, because they are most often not formalized

[8] For instance, the open-source TMT model at `https://github.com/Open-MBEE/TMT-SysML-Model`

[9] Effectively here refers to the so-called "cognitive effectiveness": the speed, ease and accuracy with which a representation can be processed by the human mind. [70]

in any way. So when UML or SysML diagrams are used for anything more than informal communication between humans, these hidden semantics may be interpreted wrongly, or they may simply be lost. Several SysML models depicted in publications about MBSE in telescope modeling (e.g. in [53], [17] and [21]) show that varying symbol colors, positions and sizes are frequently used in practice. So while these diagrams may sometimes be "worth ten thousand words" when they are used for informal communication [57], they may cause false interpretations or loss of information otherwise.

A final issue about graphical modeling is about productivity. Is drawing a graphical model any faster than writing an equivalent model in some text-based language? Does it require less effort to navigate through a graphical model, to modify it, to extract information from it? Much of the answers to these questions depend on the user-friendliness of the tools and technologies that are used. But it is obvious that the SysML models that are currently being developed for the next generation telescopes depend on "heavy" feature-rich graphical CASE[10] tools for drawing and editing diagrams, while text-based models at the very minimum only require a text editor. Also serializing graphical models is much more complex since not only the explicitly modeled information must be encoded, but also the lay-out of the models. The serializations of UML and SysML (in XMI, the XML Metadata Interchange format) are therefore much more verbose and complex than comparable serializations of text-based models that don't need to encode any "visual" information. Furthermore, since UML and SysML semantics are not well defined, the former SysML models cannot be reliably interchanged between programs of different vendors. It means that there is a high risk of vendor lock-in, and an associated high risk that any gains in productivity in the present can quickly be lost in the future, when modeling requirements change and the tool does not comply anymore.

1.4.3 Object-oriented design

The last problem discussed in this chapter is about the object-oriented design patterns that are still prevalent in current system designs. In this section, we will argue that object-oriented design can be very useful (it is just a form of abstraction), but it may severely restrict reusability when it is applied too frequently, or when it is applied in the same (much more narrow) sense as *object-oriented programming*.

As shown earlier in 1.3.1, the need for concurrent execution models is already forcing current practices to adopt new (non-object-oriented) paradigms: for instance the TMT intends to develop actor-oriented software for controlling the main interactions within the observatory [104]. It can also be argued that the internal block diagrams of SysML (with "blocks" and "flow ports") frequently seen in telescope modeling, are closely related to actor models – albeit with the important remark that SysML only defines the syntax of these diagrams, not their execution semantics [59]! Despite these efforts, several architectural

[10] Computer Aided Software Engineering

design choices remain object-oriented even if they should not be, and therefore they remain of limited reusability (see below).

Object-oriented design tries to represent the world as interacting "objects", having state, behavior and identity according to Booch [8]. Booch further states that objects provide "a crisp boundary around a single abstraction", with an abstraction being "the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries relative to the perspective of the viewer". The last part of this definition is a very important one: object-oriented design is all about creating *boundaries*, as seen from a *particular point of view*. While models always materialize a certain interpretation of reality, the act of "creating boundaries" is a very artificial one. This clearly applies to embedded systems (and even more so, cyber-physical systems) which are, by definition, tightly integrated with their environment – an observation which is opposite to having "crisp boundaries". An object-oriented model thus not only represents a subjective view of the world, but also a much over-constrained one. The more it is applied, the more the model becomes over-constrained.

The goal of object-oriented design is to simplify a problem, by reducing the world to objects that have crisp boundaries (or interfaces or "contracts" [69]) through which they can interact. As long as the constraints imposed by the model hold (i.e. as long as the boundaries or "contracts" remain unchanged), the object model offers several mechanisms (abstraction, encapsulation, modularity and hierarchy [8]) that can be used to an advantage or to a disadvantage. Problems start to occur when the constraints are violated, and the object-oriented model turns out to be an *over*-simplification of reality. This is very natural for the development and maintenance of embedded systems, because these processes involve multiple engineering disciplines and hence multiple interpretations, because requirements tend to change over time, because technology evolves and must be replaced, etc. Unfortunately, object-oriented design often leads to "coarse-grained" design in practice, with objects having artificially large boundaries. By making too many assumptions about the subjects being modeled, these designs violate the "single responsibility principle" (as it is called in sofware development [65]) and therefore cannot easily evolve. This is aggravated by the frequent use in object-oriented programming of *subsumption* (the "is-a" relationship) as an abstraction mechanism. Subsumption is a very "tight" way of coupling objects, as it forces all objects of a certain type to inherit *all* properties of another type of objects. The slightest change in the parent type may have ramifications for all descendant types. Fortunately, the tight coupling of the subsumption relationship can often be relaxed by using the composition ("has-a") relationship, as will be shown with an example in the next chapter (see 2.2.1).

Looking at the previously described current practices, it appears that the artificial boundaries of object-oriented design are still very present in current telescope control systems. One example is how control systems are broken down into components and layered in several tiers, each separated by a "crisp"

interface. As mentioned in 1.3.1, clearly this model restricts the control system to a hierarchical control flow (flowing from top to bottom), which makes it hard to satisfy the concurrency requirements of the next generation of telescopes. But what's more is that the boundary of a component or a layer (or an object in general) also hides all implementation details. *Information hiding* is an essential part of object-oriented programming, but it explicitly makes information unavailable for reuse by others or for reuse in the future. The same remark applies to the current practice of "integrating" off-the-shelf components by *encapsulating* them, e.g. via a wrapper library or a proxy server that "map" some state and behavioral information between the component and the rest of the system. By considering the off-the-shelf component as a black box with an interface, reuse of information (across the boundary of the mapping) becomes very limited. In addition, the mapping itself is a very obvious "artificial" addition to the system. It is an artifact that must be developed and maintained, and that may cause additional problems (e.g. it may introduce time delays, additional traffic on the network, etc.). Finally, not only the technology itself but also the "problem" becomes encapsulated: engineers on each side of the mapping are stimulated to fulfill their traditional roles and use traditional methods, instead of finding commonalities and other ways to homogenize the overall design and workflow (e.g. by using similar models, naming conventions, version control tools, etc.).

1.5 Research goal

The goal of this thesis is to work out a solution that addresses the problems of the previous section, in way that can be realized today. As we will argue in the next chapter, such a solution requires a fundamental methodology change, because the identified problems are inherently bound to the methodology behind the current practices.

More in detail, our goal is therefore to:

1. elaborate and derive the requirements for such a methodology change;
   → *see chapter 2 "Methodology"*
2. implement a framework that satisfies these requirements;
   → *see chapter 3 "Implementation"*
3. apply this framework to a real telescope;
   → *see chapter 4 "Application"*
4. evaluate the framework by analyzing the implementation and application;
   → *see chapter 5 "Evaluation"*
5. conclude by reasoning about a generalization of the results.
   → *see chapter 6 "Conclusion"*

In the above points, the word *framework* is used in its general sense of "the basic structure of something: a set of ideas or facts that provide support for something" [68]. It's an abstraction of the solution that we propose, highlighting ideas that differentiate our solution from others.

An explicit deliverable of the PhD project is a new control system for the Mercator telescope: a 1.2 m optical telescope located at the Roque de los Muchachos Observatory on the island of La Palma (Canary islands, Spain). It is the subject of the application chapter, acting as a proof-of-concept for the framework. Much more than a temporary test set-up in the "ideal" environment of a laboratory, such a real-world application reveals the true implications of the proposed methodology change. For instance, in case of the Mercator telescope, the framework must be able to deal with incomplete knowledge about legacy devices, with changing requirements due to feedback of the local staff and observers as they gain experience with the new system, with hardware failures that inevitably occur while the telescope is operated during 360 nights per year, with "special" requirements for rarely used functionality, and so on.

The particular implementation of this framework is naturally tailored to the application. For instance, the choice to implement detailed software modeling and to mostly leave out mechanical modeling is based on the pragmatic reasoning that the software of the Mercator telescope must be fully replaced, while the mechanics are already in place. So while the implementation of the framework is tailored to the Mercator telescope, the design of the framework – and the lessons that we learned by our particular implementation – is much more reusable.

In this thesis, related work is referenced throughout the chapters, whenever relevant. We described several subjects of the thesis in earlier publications, more specifically:

- the design of the Mercator telescope control system, described in 4.1, has been first laid out in 2010 in [86], and has been reviewed in 2016 in [87];
- some subsystems of the telescope, described in 4.3, were elaborated earlier in [80] (hydraulics subsystems, 2008) and [83] (tertiary mirror subsystem, 2012);
- our choice for the OPC UA communication technology, as described in chapter 4, was based on [82] (2011) and [84] (2012);
- our migration from informal to formal modeling, described throughout this thesis, has already been documented in publications [83] (2012), [88] (2013), [85] (2014), [89] (2015), and [87] (2016);
- general thoughts about the opportunities of semantic models in industrial engineering, some of them described in chapter 6, were already published in 2015 in [81] and in 2016 in [110].

1.6 Summary

This chapter started by defining the subject of this thesis: telescope control systems. Telescope control systems are examples of *distributed real-time embedded (DRE)* systems, a designation which highlights the tight interaction between the system and its environment. As DRE systems advance towards future "intelligent" cyber-physical systems (CPS), the boundaries between the computational and the physical processes are expected to fade even more, making it vital for *knowledge* about the system and its environment to be available. Looking at the evolution of telescope aperture over the last four centuries, it is clear that the size of telescopes and their control systems is increasing, lately perhaps even faster than before. Increasing size leads to increasing complexity (sometimes at the same rate, sometimes less, seldom more), increasing development and maintenance costs, and increasing importance of qualities such as reusability, traceability, verifiability and evolvability. These qualities are pursued by the current practices of developing software frameworks and systems engineering models, and of integrating off-the-shelf components. While these current practices are mature, trusted, and capable of delivering even the largest telescopes ever built, three problems were identified that remain unaddressed. Firstly, current practices are based on informal knowledge representation, which makes models imprecise and over-constrained at the same time. Secondly, they use a graphical notation which adds even more informality to the models. Finally, their frequent use of object-oriented modeling patterns results in an over-simplified representation of reality, and thereby compromises the reusability and evolvability of the designs.

Chapter 2

Methodology

LOOKING back at the current practices of modern telescope control systems as elaborated in section 1.3, a commonality can be identified. The starting point for the development strategies behind the telescope control systems of today is the *modeling* of the actual systems. The efforts appear to concentrate very much on modeling the system requirements, structure and behavior, ... all within the frameworks of traditional modeling languages and modeling paradigms. Whether they represent a three-tiered component-based heterogeneous software framework or a SysML-based system design, the models are developed according to popular and mature practices, and are literally the result of a pursued "model-driven" development methodology.

While recognizing the reasons for applying these current practices to existing observatories or those already under development, an analysis of the current practices reveals some fundamental problems. As shown in section 1.4, the informality of the used modeling languages, their graphical notation, and the prevalence of object-oriented modeling patterns are significant drawbacks of the current practices. We consider these problems as "fundamental" because they are over-constraining the designs, making it very hard to apply solutions in the future, as the projects evolve through their projected lifetime of several decades. They reduce existing models to rather arbitrary "assemblies" of pieces of information, without proper meaning, without proper context. As a result, they compromise the reusability of the models, making it hard to reuse information across the boundaries of engineering disciplines, across the boundaries of technologies, and across the phases of the lifetime of the project.

Because current model-driven development methodologies lead to informal and over-constrained models, and because such models compromise the reusability of information, a methodology change is needed. A new methodology will therefore be proposed in this chapter, and its requirements will be elaborated. Subsequent chapters will describe an implementation that tries to satisfy these requirements, followed by the application and finally verification of the methodology.

2.1 From model-driven to knowledge-driven

Currently developed models of telescope control systems can be positioned on the so-called Data-Information-Knowledge-Wisdom (DIKW) hierarchy or "knowledge pyramid" [2] [94]. Definitions in literature of *data*, *information*, *knowledge*, and *wisdom* vary greatly, are often contradictory and sometimes even the subject of philosophical debate [94]. Below we therefore give a personal and pragmatical interpretation of the DIKW hierarchy applied to modern telescope control systems, as explained using the transmission ratio example of the previous chapter (see 1.2.3).

- **Data** are the "bits and pieces" of information that are used to build the control system. They are facts and observations that need to be organized and processed to become useful. An example is the observation that a particular "transmission ratio equals 1 : 1440".
- **Information** consists of aggregated, organized and processed data. An example would be a SysML model of an azimuth axis, consisting of a motor with a maximum velocity of 15 000 rpm and a reduction with a transmission ratio of 1 : 1440. This information is useful because it is organized in such a way that it can be reused (e.g. to generate documentation) – if it is interpreted as the author of the model intended! For instance, without further context, it is unclear if the "maximum velocity´´ of a DC motor should be interpreted as the motor's no-load velocity, or as its maximum rated velocity for continuous operation.
- **Knowledge** adds information *about* this information, so that the information becomes meaningful. By adding context, it becomes possible to distinguish a transmission ratio from e.g. a mass ratio, and to calculate the maximum velocity of the azimuth axis based on the transmission ratio. Knowledge thus not only contains information about the azimuth drive system, but it also specifies how this information should be interpreted.
- **Wisdom** adds purpose and judgment to the acquired knowledge. Wisdom would be the judgment that a particular motor and reduction combination are the optimal choice for building a particular telescope. As described in literature, judgment is not just based on "rational" arguments: it requires vision, foresight and ethical judgment to appreciate why a particular motor and reduction combination is indeed *the best* choice for a particular telescope. As noted by Rowley [94], wisdom perhaps has more to do with human intuition, understanding, interpretation and actions than with systems. In reality, choosing a motor and a transmission is indeed heavily influenced by personal preference (e.g. for a particular vendor), by pragmatics (e.g. a short delivery time is considered more important than higher performance), by intuition (e.g. the maximum permissible torque must be overdimensioned by a safety factor of 3), by aesthetics, and so on.

This is visualized in figure 2.1.

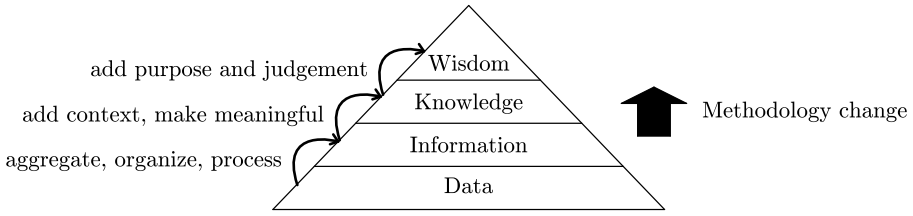


Figure 2.1: The DIKW hierarchy.

When applying the current practices and problems of telescope telescope control systems – as identified in the previous chapter – to the DIKW hierarchy, two claims can be made:

1. The current practices behind telescope control systems are only producing information, and not knowledge.
2. This information, as it is currently expressed, cannot evolve into knowledge.

The first claim is based on the informal nature of UML and SysML. Because these languages are specified informally, the meaning of SysML and UML models is ambiguous, making it impossible for the information to be interpreted correctly. The second claim is a consequence of the current practice to over-constrain the models. As described in section 1.4.1, UML and SysML over-constrain the meaning of its primitives informally, and thereby make it very difficult to reuse these primitives by domain-specific extensions. As a result, users of these languages typically disregard the standard semantics of UML and SysML and invent their own semantics – thereby affirming the positioning of the models in the 'information' layer of the DIKW hierarchy. In short: **SysML is not a knowledge representation language, and cannot evolve into one**. Extensions of UML or SysML (sometimes even called *ontologies*, as in [120]) cannot "undo" the constraints that were put informally on the UML and SysML primitives (as ontologies can only be extended *monotonically*, see 2.2.1). While it is common practice to do so, such extensions thus violate the UML and SysML specifications – even if the violations are not penalized by the CASE tools that are used to build the models. However, when using such inconsistent ontologies, one can only hope that tool vendors will not start to enforce all the informal constraints of the UML and SysML specifications – which are written out on hundreds of pages – at some point, and hope that any other (future) user of the models will also disregard them. From a pragmatic point of view, these risks can be weighted against the advantages of being able to use an off-the-shelf CASE tool. As we will elaborate in this thesis, we feel that the advantages of such CASE tools, and of a graphical language without proper meaning, are very limited – at least for the application described in chapter 4.

Similarly, the artificial boundaries imposed by object-oriented design do not add meaning to the models: they merely structure (and over-simplify) data to produce organized information rather than knowledge. The methodology change promoted by this thesis is to shift from *model-driven* development in its current form towards *knowledge-driven* development. It requires the ability to represent knowledge instead of information. As shown earlier, the current practices (involving SysML and UML informal semantics, graphical notation, and object-oriented modeling) are unable to meet this requirement. Therefore, a new framework will be developed. This framework must be capable not only of representing (some of) the knowledge of telescope control systems, but it must also be capable of reusing this knowledge to improve the qualities of the systems. We will derive the requirements of such a framework in the next section.

2.2 Requirements

The "knowledge-driven" methodology advocated by this thesis proposes three changes with respect to current practices: i) the semantics of the domains of interest must be formally specified by ontologies, ii) the systems must be modeled accurately by domain-specific languages and suitable modeling paradigms, and iii) tools must be available to support modeling, reasoning and artifact generation. These three changes will be elaborated below.

2.2.1 Ontologies

The imprecise meaning of current telescope control system models is caused by the informal specification of the UML and SysML vocabulary. Thus, to be able to create models with a precise meaning, a formally specified vocabulary is required instead. Ontologies can provide such a vocabulary.

One of the most cited definitions of *ontology* is the one by Gruber [37], stating that "an ontology is an explicit specification of a conceptualization". Borst [9] found this definition to be too broad, and defined an ontology as a "formal specification of a shared conceptualization", emphasizing the need for the specification to be *formal* (machine readable) and for the conceptualization to represent a *shared* and generally accepted view. *Conceptualization* was defined earlier by Genesereth and Nilsson [32] as "the concepts and other entities that are presumed to exist in some area of interest and the relationships that hold them". Gruber [37] further noted that a conceptualization is "an abstract, simplified view of the world that we wish to represent for some purpose".

Based on the above definitions and the discussion of "what is an ontology" by Guarino et al. [39], we say that an ontology is an "explicit, formal specification of a shared conceptualization". To explain why we consider ontologies to be required for representing knowledge about telescope control systems, we will elaborate the two parts of this definition, and compare them to current practices.

An explicit, formal specification ...

According to the first part of the definition, ontologies are *explicit, formal* specifications. An ontology explicitly specifies the concepts in a particular domain, together with their definitions and their interrelationships [73]. To explain this by example, we reprise the SysML definition of the concept of *Requirement*, which was already given in the previous chapter (see 1.4.1):

> A requirement specifies a capability or condition that must (or should) be satisfied. (...) Requirements are used to establish a contract between the customer (or other stakeholder) and those responsible for designing and implementing the system.

It can be seen that the above informal definition is vague and over-constraining, but also not very explicit since concepts such as capability, condition, system, contract etc. are not defined at all by the SysML specifications. In contrast, below we try to give an imaginary, more explicit and formal version of this definition (in pseudo language). The aim of this experiment is not to create a well designed ontology, but rather to create an ontology that closely matches the informal specifications by SysML.

```
Requirement,
  Capability,
  Condition,
  Contract,
  Stakeholder,
  Designer,
  Implementer is-a Thing

specifies,
  satisfies,
  binds,
  is-satisfied-by is-a Relationship

is-satisfied-by is-inverse-of satisfies

Requirement
  Facts:
    specifies SOME (Capability OR Condition)
  Rules:
    IF (?r is-instance-of Requirement) AND (?r specifies ?x)
    THEN SOME ?y satisfies ?x

Contract
  Facts:
    has-a SOME Requirement
    binds ONLY Stakeholder
    binds MINIMUM 2 Stakeholder
...
```

Listing 2.1: Imaginary formal specification of *Requirement* by SysML.

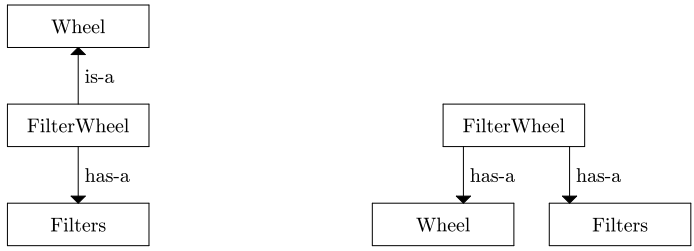
Comparison of the above informal and formal definitions reveals how problematic it is to interpret an informal specification in the way that was intended by the authors. For instance, in agreement with the SysML specifications, a

requirement may be related to a capability or condition via a `specifies` relationship. However, in the remainder of the SysML specifications there is no relationship called `specify`, and requirements are said to be satisfiable, suggesting that a requirement "is-a" capability or condition. Additionally, in contrast to our formal version, the SysML specification does not clarify if a requirement specifies "at least one" or "exactly one" capability or requirement. Further confusion arises from informal phrasing such as "must (or should)", which do not take a clear position on whether or not a model is inconsistent if the specified capability or condition of a requirement is not satisfied. Finally, the act of establishing a contract is a typical example of an n-ary relationship which is very common in real life (and very easy to express in natural language), but which can be modeled in various ways and therefore leads to ambiguity – unless all concepts and relationships are made very explicit.

While the informal specification of *Requirement* has been expressed in natural language, the formal specification by listing 2.1 has been expressed in an (imaginary) formal knowledge representation language. Formal languages provide a much smaller set of primitives than natural languages, but their semantics (i.e. the meaning of the primitives) and syntax (i.e. the way how the primitives can be arranged) are defined much more precisely. For instance, listing 2.1 represents some knowledge about *Requirement* only if the primitives `is-a`, `is-instance-of`, `is-inverse-of`, `Thing`, `Relationship`, `SOME`, `MINIMUM`, `AND`, `OR`, `IF`, `THEN` have "precisely defined" semantics, and if they are arranged and structured in a valid way. To be able to represent knowledge, such "precisely defined" semantics can be borrowed from logic-based languages, frame-based languages, rule-based languages, etc. [31]. For instance, the primitives of listing 2.1 can be based on:

- propositional logic (e.g. the operators `AND` and `OR`);
- first-order logic (e.g. the existential quantifier `SOME` and the universal quantifier `ONLY`);
- description logic (e.g. the cardinality constraint `MINIMUM` and relationship inversion `is-inverse-of`);
- frames representation (e.g. the frame `Requirement` has slots `Facts` and `Rules`);
- rules representation (e.g. the `IF THEN` rule of `Requirement`).

When structuring an ontology using frames, one must be careful to avoid the pitfalls of object-oriented design. Frames represent the world as sets and subsets of concepts, similar to classes and subclasses of object-oriented design. Their goals are different however: frame-based representations primarily aim to structure knowledge in a "clean" taxonomy (in order to simplify the organization of knowledge), whereas object-oriented design is more targeted at the encapsulation of related data and behavior (in order to simplify the interaction between objects). Nevertheless, both approaches provide the



(a) Definition based on `is-a`. (b) Definition based on `has-a`.

Figure 2.2: Example of definitions based on the `is-a` and `has-a` relationships.

subsumption relationship (`is-a`) and composition relationship (`has-a`). The former introduces a much greater dependency, but may often be avoided. For instance, we can model a filter wheel (a device frequently used in astronomical instruments, to place light-transmitting filters, mounted in a rotatable disk, inside the beam of light) in two ways, as shown in figure 2.2. While the left part of the figure models a filter wheel as *"a kind of wheel that has filters"*, the right part of the figure considers a filter wheel as *"something that has a wheel and filters"*. Intuitively, the former seems a correct representation of reality, and is easy to implement in software. Unfortunately it is less evolvable: what if the definition of a wheel changes? For instance, we could add that a wheel is round and has a circumference. According to the left part of the figure, our filter wheel now suddenly is round and has a circumference... which is not what we originally intended, since we considered a filter wheel to be a device (e.g. consisting of a bracket, a wheel, a motor, etc.). The right part of the figure is still valid: this definition of "filter wheel" has not changed, only the definition of one of its parts has. So while the `is-a` semantics are very strong (because they infer that *all* properties of the base type(s) are inherited), the `has-a` semantics are much more weak. The strong semantics of the `is-a` relationship can cause very "rigid" (little evolvable) definitions, especially if deep subsumption hierarchies are created, or multiple base types are subsumed. The latter is less a problem for frames than it is for object-oriented classes (e.g. the "diamond problem"[1] is only caused by multiple inheritance in object-oriented programming), but in both cases problems may occur if the definition of one of the base types change. The `has-a` relationship is therefore often a safer choice than the `is-a` relationship, and should be favored if possible.

More in general, a formal language with well defined semantics and syntax allows reasoning: logical conclusions can be derived (inferred) from the statements expressed with this language, by a computer program. If more primitives are added to the language, then the language becomes more expressive, but

[1]The diamond problem in object-oriented programming is the ambiguity that arises if a class D inherits from two classes B and C that override a common base class A. In this case, if a method defined by A should be called, then it is unclear whether to call the overridden method by B or by C.

harder to reason about. For instance, natural language is very hard to be understood by machines not only because of the ambiguous semantics of many words, but also because of the huge number of words and combinations of words that can be used to express something. Ideally, a knowledge representation language should both be expressive enough to allow structuring and encoding all the necessary knowledge easily, and to make the represented knowledge understandable by humans [31]. However, languages can easily become very hard or even impossible to reason about, if too much expressiveness is added. Even very simple logic-based based languages can be *undecidable*, meaning that some statements expressed in this language cannot be proven despite the logical foundations of the language. For instance, it is impossible for the compiler of a programming language to find out if an arbitrary computer program and input can finish in finite time or will run forever[2] – despite the limited expressiveness that is needed to describe a computer program (Turing machine). Thus, if we expect the ontology of listing 2.1 to serve for classifying (or other reasoning about) the requirements, contracts, stakeholders etc. of *any* (!) system that is modeled using the ontology, then the logic on which our imaginary language is based must be decidable. More expressiveness can be added, but then only a restricted set of models expressed using the ontology can lead to provable conclusions, and no "general" conclusions can be drawn. For instance, if the language is sufficiently expressive to model simple mathematical formulas, it may be required to restrict the values of the mathematical operands to given values, for the model to be decidable.

... of a shared conceptualization

The second part of the definition of ontologies is not about *how* an ontology should be specified, but about *what* should be specified. What are the shared "concepts and other entities that are presumed to exist in some area of interest, and the relationships that hold them" [32] that we should model?

First of all, it should be noted that the answer to this question is restricted by the choice of knowledge representation language. Not only the expressiveness of a particular knowledge representation language, but also the type of language affects the knowledge that can be expressed. For instance, frame-based languages lead to a structured view about the world by defining frames (which are similar to the classes in object-oriented design) whereas rule-based languages describe properties of the world as an unordered set of if-then rules. Frame-based and rule-based representations may also be combined, so that rules about a specific concept can be tied to the frame that represents this concept. Choosing a representation language thus inevitably affects which knowledge can (and which cannot) be represented.

Knowing the limitations of what knowledge *can* be expressed by a particular language, the question remains of which knowledge *should* be expressed. Several

[2]This decidability problem was originally described by Turing in 1936 [106] and has later become known as the *halting problem*.

methodologies and guidelines for designing ontologies are described in literature: see for instance [38], [27], [76], [107] and [55]. Based on this literature, we have listed a set of properties that we find required (or at least highly desirable) for our use case of designing telescope control systems.

1. **Adequate purpose and scope.** Knowledge about telescope control systems spans various domains, so we expect to create multiple ontologies whose purpose and scope must be carefully defined. To foster reuse and knowledge sharing, Neches et al. argued that the ontologies should consist of layers of increasingly specialized, less reusable knowledge [74]. While we think that the idea of creating artificial and explicit "layers" should be avoided if possible (because it may hamper reusability much like object-oriented design does), we agree on the need to create a range of ontologies, from highly abstract ontologies to highly domain-specific ontologies. An approach to modularize the knowledge can be adopted from [107], suggesting a *middle-out* analysis (rather than top-down or bottom-up) because such an approach balances the need for modeling detail against the need for modeling commonalities. The same idea can be applied to each individual ontology: by first defining the most important concepts of a domain in fairly abstract terms, it becomes easier to agree on a shared conceptualization among domain experts while maintaining stability as both details and commonalities are discovered and added at a later stage.

2. **Rigorous formality.** To avoid the problems caused by informal specification, we require all knowledge to be declared *rigorously formal*, which Uschold and Gruninger [107] defined as "meticulously defined terms with formal semantics, theorems and proofs of such properties as soundness and completeness." It means that natural language descriptions, if present in an ontology, should not be used to infer anything nor to verify the consistency of a model. Another consequence is that **specific concepts must be highly constrained.** For instance, to be able to generate Python software code, we need a very specific Python ontology which can be mapped one-to-one to Python code. A one-to-one mapping is clear and does not require a possibly confusing description in natural language. This approach avoids the ambiguity that occurs when, for instance, an informally described "general" UML class diagram is converted to "specific" Python code by some mapping implemented by a particular tool. The existence of a Python ontology does however not necessarily imply that a software developer needs to encode his/her knowledge *directly* in the Python ontology. If a more abstract software ontology is available, and the Python ontology is explicitly and formally mapped to this ontology, then a software developer can encode his/her knowledge using the abstract software ontology and let the reasoner infer the corresponding Python concepts. The important point is that, in this case, both the mapping and the Python ontology are not "encapsulated": they are explicitly and formally described, and thus part of the knowledge base.

3. **High extensibility.** In contrast to very specific ontologies, abstract ontologies must be extensible. To quote Gruber [37]: such ontologies should offer a conceptual foundation for a range of anticipated tasks, and the representation should be crafted so that one can extend and specialize the ontology *monotonically* (i.e. they should allow the definition of new terms based on the existing vocabulary, without requiring a revision of the existing definitions). Still according to Gruber, an important design principle is therefore that of minimal ontological commitment: an ontology should make as few claims as possible about the world being modeled, to support the intended knowledge sharing activities. Since this "minimal" ontological commitment rises to the maximum possible commitment for very specific concepts (because it must be possible to map them one-to-one to real world concepts), we agree with Borst [9] that ontologies should rather have the **right ontological commitment**. Only those concepts should be defined whose differences can be expressed with the given knowledge representation language. Compared to UML and SysML, it means that abstract concepts will make much less claims about the world, making them much more reusable.

4. **High clarity, concision, coherence, verifiability.** The ontologies should have high clarity: they should effectively communicate the intended meaning of the terms, unambiguously and objectively, independent of social or computational context [38] [107]. They should also be concise: they should only define relevant terms [27], in agreement with the intended purpose and scope of the ontology. Ontologies should also be coherent (i.e. they should not define unsatisfiable concepts [29]) and should enable verification (i.e. they should detect when a system model is inconsistent with the ontology definitions).

Shared conceptualizations as defined above should offer a common terminology and a common understanding to describe the actual telescope control systems. They can be used to create a new set of languages to describe those particular systems, but they are much more semantically "rich" (expressive and meaningful) compared to their UML and SysML counterparts. What's still missing is a notation, as we will elaborate in the next subsection.

2.2.2 Domain-specific languages

Describing actual systems is very different from describing the domain concepts that are needed to describe the actual systems. In our framework, domain concepts are described by *ontologies* using *knowledge representation languages*, while actual systems are described by *system models* using a set of *domain-specific languages (DSLs)*. These DSLs reuse the terms of the ontologies (and their semantics), and additionally define a notation. The requirements for this notation will be elaborated below.

Fowler defines domain-specific languages as "computer programming languages of limited expressiveness focused on a particular domain" [64]. Thus, they should be formal, concise, and about a particular domain, similar to ontologies. Unlike ontologies however, they are used to describe a system instead of the domain knowledge related to that system. Still according to Fowler, a DSL is a thin veneer or *facade* over an underlying "model", and it's important to separate the benefits provided by the model from the benefits of the DSL. In our framework, this "model" of the real world is represented by a set of ontologies, so we can state that a **DSL adds a facade over an ontology**. Since the semantics of the DSL primitives are already provided by the ontology, it means that the facade must still provide: i) a mapping between the language primitives and the ontology terms, and ii) a syntax for the language.

The mapping between the language and the ontology can be very simple: if the terms defined by the ontology are valid names according to the DSL syntax, then these terms can even be reused without modification. If not, then a conversion algorithm (or even a manual mapping) must be defined instead. For instance, a DSL based on the exemplary ontology of listing 2.1 may be able to reuse the terms `Requirement`, `Contract` and `specifies`, but may be unable to reuse the terms `is-a` and `is-satisfied-by` unless replacing the hyphen characters because (as with many programming languages) the "-" symbol would be treated as the mathematical minus operator.

A more difficult exercise is the definition of a syntax, since this syntax accounts for most of the added value of a DSL. Its requirements are described below. They are derived from the shortcomings of current practices (see 1.4) and the opportunities and risks of DSLs described in literature (see [64], [109], [67]).

1. **Textual notation.** Because graphical notation can only convey a limited amount of information, because it inherently conveys informal information, and because it depends on heavy tools, we require the notation of the DSL to be textual.

2. **Rigorous syntactic rules.** For a DSL expression to be meaningful, it must be possible to unambiguously convert the DSL expression to an expression written in the knowledge representation language of the underlying ontologies (i.e. those behind the facade of the DSL). It implies that syntactic rules must rigorously decide whether or not a DSL expression is valid, and how a valid expression should be "translated" to an expression in the knowledge representation language of the underlying ontologies.

3. **High clarity and concision.** If syntactic rules are more clear (e.g. because they are simple and well documented) and more concise (e.g. because few punctuation and other symbols are needed to express something), then DSL descriptions are easier to read, to write, to modify, to debug, and to communicate. This improves productivity since less human effort is needed.

Syntactic rules that are specifically targeted at making a language more easy to read or to express (sometimes called "syntactic sugar") can make a big contribution to make a DSL more productive. For instance, the imaginary language of listing 2.1 is pretty clear and concise because no special delimiters (other than whitespace) are needed to separate statements, and because single characters may replace lengthy partial statements (e.g. each comma of lines 1–6 replaces the partial statement `is-a Thing`). More syntactic sugar may be needed however to efficiently describe a complex embedded system. For instance, complex systems are often modeled as systems of systems of many levels deep (e.g. a mechanical design may be structured as an assembly of assemblies of assemblies and so on). A recursive syntax (e.g. with increasing indentation for increasing recursion depth) may be able to express such "recursive semantics" much more clearly and concisely. So while a DSL notation should not introduce new semantics, it should enable humans to reuse the semantics of the underlying ontologies more efficiently.

A simple text editor – or even a pen and paper – is all what's needed to encode a system model using the DSLs described above, and to communicate this model among people. But, as will be elaborated in the next subsection, the benefits of the system model and the underpinning ontologies can only be realized if tools are available that can actually *do something* with them.

2.2.3 Tool support

Current practices use tools to draw SysML and UML models, to syntactically verify those models, and to generate artifacts such as source code and documentation (ranging from diagrams to be included in documents, to complete documents such as interface control documents). Verifying these models at a semantic level is currently limited to a few specific use cases such as state analysis [52] and simulation of the worst-case duration of a process [51], all of which depend on domain-specific extensions of UML. As elaborated in 1.4.1, these extensions can only be tool-specific (as their underlying UML and SysML semantics are informally specified) and disparate from other extensions (as SysML and UML concepts are too constrained to serve as an abstract "foundation" for multiple domain-specific extensions). So while the effectiveness and the potential benefits of the currently used tools are much restricted by the lack of formality of UML and SysML, their design goals are clear: they want to assist model creation and manipulation, verification at a syntactic *and* at a semantic level, and the generation of all kinds of artifacts. Based on these goals, we can specify the functional requirements of our framework:

1. **DSL syntax verification and mapping.** A DSL only creates a facade over an ontology, by defining a syntax (i.e. a set of well-formedness rules) and a mapping between the DSL and the ontology. Hence, tool support should be available to write a syntactically valid DSL script (by verifying those well-formedness rules, preferably on-the-fly), and to map

the DSL expressions to expressions written in the knowledge representation language of the ontology behind the facade.

2. **Reasoning.** Tools should support reasoning, i.e. they should be capable of inferring the logical consequences from the asserted facts of the DSL models, based on the inference rules of the ontologies. Reasoning should allow the discovery of new relations: for instance, an abstract software class model named "My-class" could be automatically classified as a Python class model if Python is the target language of the software project according to the model. Reasoning also allows the verification of the DSL descriptions: for instance, the Python ontology may infer that the newly classified "My-class" Python class is inconsistent with the rule saying that hyphens are not permitted in Python names.

3. **Artifact generation.** With "artifacts" we mean any kind of output that can be generated by software, including web pages, printable text documents, figures, configuration files, source code, and so on. The idea is that instead of encoding design knowledge directly in these artifacts, it is better to encode this knowledge in models and then "transform" this knowledge into artifacts by a software procedure. In this way, the knowledge becomes more reusable (and hence more consistent, traceable, verifiable, and evolvable, as discussed in 1.2.3). Likewise, the software procedure to transform the knowledge is reusable, so creating, debugging and maintaining this software procedure may be much less costly than creating, debugging and maintaining the artifacts manually. The ability of third-party tools to reuse (or "import") generated artifacts may be an important criterium to select the tools and technologies for building a telescope control system. The selected technologies and tools must be *model-friendly*: it must be possible to encode some knowledge based on a selected technology (e.g. the configuration of a network based on a particular communication protocol, or the software of a system based on a particular programming language) in a format that can be imported by third-party tools.

Apart from the above functional requirements of the tools, also several non-functional requirements (or more specifically the "qualities" as we called them in 1.2.3) can be listed. Many of the "-ilities" (availability, extensibility, maintainability, reliability, portability, testability, etc.) are highly desirable for any software project, but we focus on one in particular: **usability**. Usability is the "appropriateness to a purpose" [10], in this case the appropriateness to the development of a telescope control system. Usability is about effectiveness ("Is it the right tool to use?"), efficiency ("How much resources are saved by using the tool?") and satisfaction ("How happy are the end-users to use the tool?"). It's an important quality because the cross-domain nature of embedded systems brings together domain experts with very different backgrounds, and all these domain experts will have to use the tool in some way or another to realize the added value of the methodology proposed in this dissertation. It means that efforts must be spent on making the tools usable by domain experts who often

are not familiar with ontologies, DSLs, inference rules, etc. The fact that the methodology has been applied to a *real* telescope (and not just a restricted test set-up), with *real* stakeholders who have to support the telescope for 360 nights per year for several years to come, makes usability especially important.

2.3 Summary

This chapter started by classifying the currently developed models of telescope control systems as *information*: structured and processed but otherwise meaningless data. To deal with the problems identified by chapter 1, this information must be augmented by domain information to become *knowledge*. Unfortunately, because the current models are expressed in an informal way, the currently applied model-driven development approaches cannot evolve into knowledge-driven development approaches. Therefore, a new framework must be developed, the requirements of which are identified below. We define three main requirements (R1, R2, R3), and a number of "subrequirements" that are derived from them:

R1 *Ontologies shall formally specify the semantics of the domains of interest for the development of telescope control systems.*

- **R1.1** *The ontologies shall have an adequate purpose and scope.*
- **R1.2** *The ontologies shall have rigorous formality.*
- **R1.3** *The ontologies shall have high extensibility.*
- **R1.4** *The ontologies shall have high clarity, concision, coherence and verifiability.*

R2 *Domain-specific languages (DSLs) shall be available to model the actual telescope control systems.*

- **R2.1** *DSLs shall have a textual notation.*
- **R2.2** *DSLs shall have rigorous syntax rules.*
- **R2.3** *DSLs shall have high clarity and concision.*

R3 *Tools shall be available to support the knowledge-driven methodology applied to the development of telescope control systems.*

- **R3.1** *Tools shall support DSL syntax verification and mapping.*
- **R3.2** *Tools shall support reasoning.*
- **R3.3** *Tools shall support artifact generation.*
- **R3.4** *Tools shall have high usability.*

In the next chapter, a framework will be implemented that attempts to satisfy these requirements.

Chapter 3

Implementation

The abstract requirements that were laid out in the previous chapter, will be broken down into more specific requirements in this chapter, and an implementation will be proposed. The wisdom behind the process of increasingly detailing and constraining the framework design is, much like telescope control system design, not just based on rationale (i.e. objective knowledge). Very often, less-than-optimal choices are made for pragmatic reasons. For instance, tools and technologies may be selected because they are available, because they can be adapted and deployed within the time frame of the PhD research project, because they can be supported more easily by the stakeholders of the application of chapter 4 to which the framework is applied, and so on. Conform to the definition of "wisdom" in section 2.1, the framework development in this chapter is thus biased towards a specific purpose – in this case, building a new control system for the Mercator telescope by applying a new and more broadly applicable methodology, within the time constraints of a PhD project. It is the reason why the requirements of the previous chapter were very abstract, because a higher level of abstraction implies a higher reusability by projects with a different purpose.

3.1 Framework architecture

The framework architecture outlined in this section offers a highly abstract view of how the ontologies, system models and artifacts – all of which were introduced in the previous chapter – fit into the framework and how they are related to each other. In agreement with the definitions below, the framework architecture is a *model* of our framework because it represents the framework for the purpose of understanding how its components influence each other. Architectural models of frameworks in literature are confounded by varying interpretations of terms such as *system*, *model*, *metamodel*, *instance of*, *conforms to*, *represents*, etc. Hence, we first define these terms before fitting them into an

architectural model, and before correlating them with the previously introduced terms such as ontologies, DSL scripts, produced artifacts, etc.

3.1.1 Definitions

- A **system** is something that is composed of elements which are directly or indirectly related to each other.
- A **model** is something that represents a system, for a specific purpose.
- A **metamodel** is something that explicitly specifies a model.
- A **metametamodel** is something that explicitly specifies a metamodel.
- **To represent** means to act as a substitute for, to take the place of in some respect, to stand as an equivalent of.
- **To specify** means to name or mention exactly and clearly, to be specific about.

We first note that in the above definitions, we intentionally stated for example that "x is something that represents y" or "x is something that is composed of y", instead of stating that "x is a representation of y" and "x is a composition of y" respectively. The latter wording is more common in literature, but it unnecessarily introduces new nouns without added value: a *representation* is naturally "something that represents something" and a *composition* is "something that is composed of something". So the semantics are really captured by the relationships (i.e. *represents* and *is composed of*), not by the nouns. Therefore, for clarity and conciseness, we simply did not introduce those nouns in the definitions.

Our definition of **system** corresponds to the one by Ackoff [1], who further clarifies that i) a system is therefore composed of at least two elements and a relation between each of its elements and at least one other element in the set, ii) each element is directly or indirectly related to every other element, and ii) no subset of elements is unrelated to any other subset. The details about what exactly it means "to be composed of something", about what exactly an "element" is, about whether or not the relations are part of the system (e.g. as in [63]), and so on, are irrelevant for the purpose of our informal architectural model, so we can leave the definitions little constrained. The definition therefore represents a wide view on systems, implying that "everything is a system": from the universe, to the telescope that observes the universe, to the atoms that compose the telescope.

Our definition of **model** largely agrees with the definitions found in literature (e.g. by Bézivin [6], Kühne [56], Naudet [72], and Favre [26]). Central to the idea of a model is the *represents* relationship, that relates the model with the real (or imaginable) world. All of the formerly mentioned authors agree that

the representation is in fact a *simplification* (abstraction, reduction), meaning that only those aspects of the system relevant to the specific purpose of the model must be captured. As stated by Kühne [56]: any representation of a real world subject automatically implies reduction and thus can be granted model status. Therefore, a system is usually represented by a set of different models, each one capturing some specific aspects [5].

Our definition of **metamodel** mostly corresponds with the definition by Bézivin [6], whom considers a meta-model to be the explcit specification of an abstraction (a simplification). If a metamodel specifies something, it must have a vocabulary to do so. Therefore, we infer that a metamodel is a model: it represents a system (i.e. some related concepts of the real or imaginable world) for a specific purpose (i.e. to specify a model). Bézivin indeed states that a metamodel defines a set of concepts and the relations between these concepts, to be used as an abstraction filter in a particular modeling activity. It means that a metamodel does not represent a model and is thus not "a model of a model", as frequently stated in literature about model-driven engineering. Instead, a metamodel makes a model meaningful, by representing some concepts that can be reused by this model. As an analogy proposed by [6], a metamodel is not like "a painting of a painting" (i.e. a representation of a representation of reality), but instead it is a representation of some concepts, reused within the painting to represent some scenery. This idea is accepted by many authors, although terminology varies (e.g. Kühne relates a metamodel and a model by an *ontological instanceOf* relationship [56], Naudet by a *defines* relationship [72], and Fabre by saying that a model must be *a model-as-mold of another model* to qualify as a metamodel [26]).

Our definition of a **metametamodel** is easily derived from the definition of a *metamodel*; only the subject of the specification changes. It follows that metametamodels are also metamodels, and thus also models. We call the concepts they represent *metaconcepts*, because they capture very primitive characteristics about concepts such as subsumption, disjointness, conjunction, etc. Some existing metametamodels specify themselves, much similar to a compiler of a programming language that is written in the programming language itself. However, one must realize that both cases still depend on a minimal set of "primitive" metaconcepts that are supposed to "already exist" and have a very clear, unambiguous meaning.

Our definitions of **to represent** and **to specify** are summaries of the definitions by dictionaries such as Merriam-Webster [68] and Collins English Dictionary [18]. They capture the primitive knowledge that is needed to understand the former definitions of models and metamodels. So while the *represents* and *specifies* relationships are very important, we can only rely on a dictionary to give a sense of their meaning. More formal constraints about these relationships have been given in literature (e.g. by expressing whether or not they are transitive, see [56]), but this is not needed for the purpose of creating an informal architectural model of our framework.

As a final note, we mention that the relationships **instanceOf** and **conformsTo** were not needed to express the above definitions. As noted by Bézivin [5] and Kühne [56], the *instanceOf* relationship originates from object-oriented design and should not be used to differentiate a model and its metamodel. As will be seen further, in our framework, *instanceOf* is indeed not only used to relate a concept of a metamodel with an instance of a model. For instance, the unit *Ampère* is an *instanceOf* a *Unit* class, both of which are part of the same metamodel that describes quantities and units. The *conformsTo* relationship is also frequently used to relate a model and its metamodel, but we note that this relationship is stronger than simply the inverse relationship of *specifies* (i.e. *specifiedBy*). A model conforms to a metamodel if it is specified by this metamodel *and* if it abides by the rules of the metamodel. For instance, if a metamodel represents a quantity value as something that has at most one unit, then the model can only conform to the metamodel if it does not wrongfully represent quantity values with more than one unit.

A summary of the above concepts and their relations is shown in figure 3.1. According to the definitions, the figure represents "a metamodel of an architectural model of a framework" (in short: a metamodel of a framework architecture).

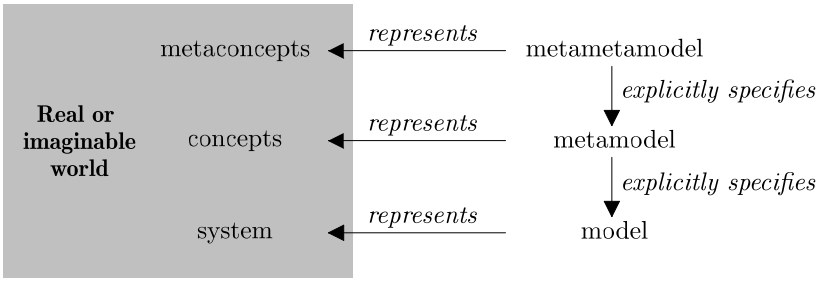


Figure 3.1: Metamodel of a framework architecture.

3.1.2 Architectural model

Now that the metamodel is clear, we can use it to specify a model of our framework. According to the definitions of 3.1.1 and the framework requirements of 2.2, we can infer that:

- **The ontologies are metamodels.** They are designed to formally and explicitly specify the system models. They represent a whole range of concepts, from abstract concepts (e.g. state machine, system) to specific concepts (e.g. requirements, telescopes, optics, ...). They are specified by a set of *knowledge representation (KR) languages* that represent primitive metaconcepts such as frames, subsumption, transitivity, disjointness, etc.

- **The system models are models specified by the ontologies.** They represent the system, or what it should look like in the future. They are written in domain-specific languages.
- **The artifacts are models of the system models.** They represent some aspects of the system model, for various purposes. As will be seen further on, artifacts can be specified by templates.

This is visualized in figure 3.2:

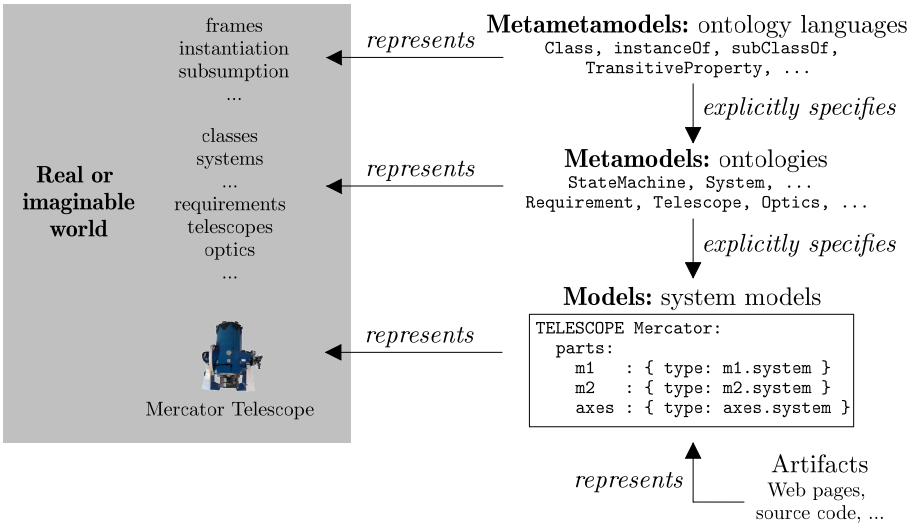


Figure 3.2: Our framework architecture according to the definitions.

3.1.3 Comparison with UML/SysML

Our definitions given in 3.1.1 can easily be applied to UML and SysML models. According to the OMG[1], UML and SysML models are part of an architecture consisting of four modeling ("M") layers:

- **M3** consists of the MOF (Meta-Object Facility): a metametamodel that is explicitly specified by itself. It represents metaconcepts such as *instantiation*, *generalization*, *association*, *multiplicity*, etc.
- **M2** consists of the UML and SysML specifications: they explicitly specify UML and SysML models at M1. The UML and SysML specifications represent concepts such as *classes*, *state machines*, *blocks*, *ports*, *requirements*, *actors*, etc.

[1]Object Management Group: `http://www.omg.org`.

- **M1** consists of UML models and SysML models. These are typically created using CASE tools by software engineers or system engineers, for a particular application. For instance, they could consist of classes such as *Telescope* and *Optics*, and instances of those classes.
- **M0** consists of the real (or imaginable) world, e.g. an existing telescope or a telescope still under study.

This is visualized in figure 3.3:

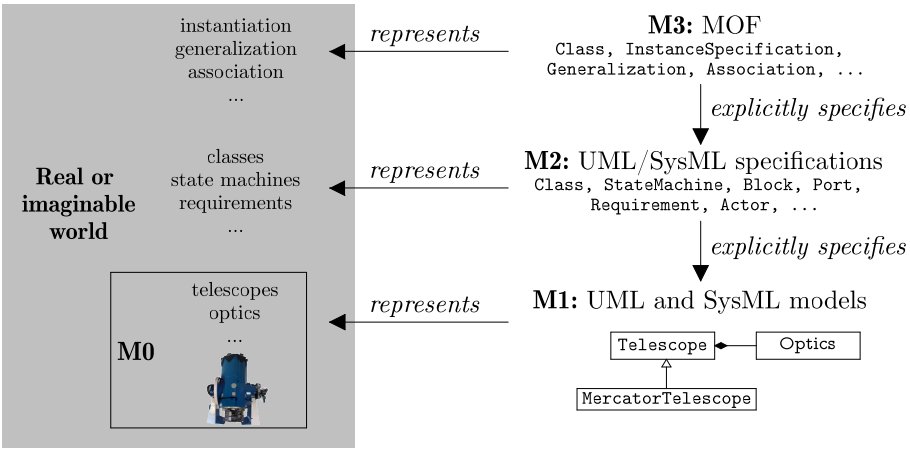


Figure 3.3: UML/SysML framework architecture according to the definitions.

At first sight, comparison of figures 3.2 and 3.3 shows that domain concepts such as *telescopes* and *optics* are represented by metamodels in our framework, and by models in the UML/SysML framework. This is not a fundamental problem because the layers of the UML/SysML framework are an arbitrary representation of reality: in theory one could create new layers below M2 to capture concepts with increasing specificity. The real problem is that, in practice, one *cannot* create new layers of increasing specificity below M2. Because, as elaborated in 1.4.1, UML tries to have it both ways: it is too specific to serve as a high-abstraction metamodel, and too abstract to serve as a low-abstraction metamodel. Another "architectural" difference is the definition of a separate layer (M0) consisting of systems in the real or imaginable world, by the UML architecture. Bézivin therefore argues that the four-layered architecture "should more precisely be named a 3+1 architecture" [5].

In the remainder of this chapter we will elaborate the components of the framework as shown in figure 3.2, and the tooling that we developed to make the framework usable for developing actual telescope control systems.

3.2 Metametamodels: knowledge representation languages

To be able to represent knowledge about telescope control systems, we need a starting point: the metaconcepts. Since we want to create formal ontologies based on these metaconcepts, we need a (set of) metametamodel(s) that very clearly and unambiguously represent them. Knowledge representation languages are designed to meet this requirement.

Several knowledge representation languages have been created in the past, including KIF, KL-ONE, LOOM, DAML+OIL, KM, RuleML, CycL, F-Logic, RDF Schema, OWL, and so on. Below are some of the characteristics in which they differ from each other:

- **semantics:** the meaning of the semantic primitives of the language, and their number, determining the expressiveness of the language;
- **syntax:** the well-formedness rules of the language, depending heavily on the representation type (e.g. description logic, first-order logic, frames, rules, ..., or a combination of those);
- **tool support:** the availability of software for reasoning about, editing, and storing ontologies;

Using these characteristics as selection criteria, we selected the following knowledge representation languages to implement our framework:

- **RDF Schema:** the Resource Description Framework (RDF) Schema, providing "mechanisms for describing groups of related resources and the relationships between those resources" [119];
- **OWL 2 RL:** a fragment of OWL 2 (the second version of the Web Ontology Language), which defines "a syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies" [116];
- **SPIN:** the SPARQL Inferencing Notation, a rule and constraint language based on SPARQL (the de-facto query language for RDF data) [54].

These languages were created to support the "Semantic Web": a vision of the future world wide web, published by Berners-Lee, Hendler and Lasilla in 2001 [4]. In this vision all information on the internet should be described and linked to each other by ontologies, instead of being embedded in web pages as natural language text that can only be consumed by human beings. The latter "web of documents", as the internet stands for today, can be read (via a web-browser), navigated (via hyperlinks) and searched (via search engines) – in natural language only. Indeed it takes a human being to enter some keywords in a search engine, to browse the many relevant and the many more irrelevant

search results, to skim through the displayed text, in order to find and aggregate the information needed to solve a task. In contrast, if the information of the Web would be described and linked by ontologies, a single query could return only the needed information in a machine understandable way, in order to solve the same task fully automatically. "Intelligent agents" could autonomously query the web, aggregate the results, reason about the results, and act accordingly.

Today, the vision of Berners-Lee of a global Semantic Web remains largely unrealized, as the many problems of dealing with such a vast, decentralized, uncertain, incomplete and inconsistent semantic network [14] remain to be solved. However, many of these problems can be avoided when Semantic Web technology is applied at a much smaller scale. For instance, within a single organization, a "miniature semantic web" may be deployed as a single central knowledge base governed by a handful of authoritative domain experts (or better called "benevolent dictators" perhaps) that rule out uncertainty and inconsistency. As will be shown in the next chapters, deploying such a miniature semantic web turns out to be feasible at least at the scale of the Mercator Telescope.

Below we will apply the previously listed knowledge representation language characteristics to RDF Schema, OWL 2 RL, and SPIN, in order to explain why we selected these languages.

3.2.1 Semantics

OWL is fundamentally object-oriented: it defines *classes* as sets of *individuals* [117]. In OWL, all individuals belong at least to one class called `owl:Thing`. Individuals can be related to each other via *object properties* or to literals (e.g. strings and integers) via *data properties*. Named *resources* such as classes, properties, and individuals with an explicitly specified name are uniquely identifiable by an IRI[2]. IRIs can be shortened to *QNames* consisting of a prefix, a colon and an identifier (e.g. `owl:Thing` is a QName of the IRI `http://www.w3.org/2002/07/owl#Thing`). IRIs do not have to be resolvable (i.e. not all ontologies can be downloaded from the internet) but since ontologies are designed to be reusable, it is considered a "best practice" to make them resolvable [42]. All ontologies (metamodels and models) developed within this thesis have been made resolvable: for instance, one can point a web-browser to `http://www.mercator.iac.es/onto/metamodels/systems` to download the *systems* ontology (see further).

OWL embraces the *open world assumption (OWA)*, saying that any statement may be true unless it is explicitly stated to be false. It embodies the notion that the knowledge about something is never complete, in agreement with our earlier definition of a model as a "reduction" of reality. For instance, if a model only explicitly states that an electric conctact `A` is connected with contact `B`, we

[2] An IRI is an "internationalized resource identifier", much like a uniform resource identifier (URI) but capable of containing non-ASCII characters such as ° and ø.

cannot simply assume that `A` is not connected to another contact `C` (unless, for instance, we add the statement that "`A` has only one connection", or the explicit statement that "`A` is not connected to `C`"). In fact, based on the statement that "`A` is an electric contact" we even cannot assume that `A` is not something completely different... such as a person, or a telescope! The OWA thus increases reusability because it constrains the world much less than the closed world assumption (CWA) – which is most often implicitly assumed for UML models. We think that this is a strong asset of OWL, since it prevents false assumptions to be made. After all, is an electric contact *always* different from a telescope (or in other words: is the class of electric contacts really fully disjoint from the class of telescopes)? Problems of electric noise traveling from telescope motors to the read-out electronics of an astronomical detector, tell us otherwise.

Another fundamental difference between OWL and UML concerns the *unique name assumption (UNA)*, saying that two individuals with a different name represent different things in the world. OWL explicitly does not adopt the UNA, meaning that a name has no unique interpretation. It also means that synonyms may exist: for instance a mechanical and an electric design may both define a system which can be made "equivalent" by adding an `owl:sameAs` object property between them. In UML on the other hand, all individuals are different, again implicitly constraining the world much more, thereby impeding reusability.

As elaborated in 1.4.3 and as illustrated by the above examples, object-oriented design can be a very useful abstraction mechanism, but it should be applied in a sensible way. Constraints on a class of individuals should only capture the essential characteristics of that class. RDF Schema and OWL offer many ways to formally constrain the meaning of classes, properties, and individuals. Below we list some of those mechanisms via two small excerpts of our *control systems* ontology (listing 3.1) and our *electricity* ontology (listing 3.1). Both listings only represent a very small part of the ontologies, written in Manchester syntax [114], for the purpose of illustrating the expressiveness of RDF Schema and OWL. The full *control systems* and *electricity* ontologies are described in 3.3.12 and 3.3.16, respectively.

```
1  Ontology <http://www.mercator.iac.es/onto/metamodels/controlsystems>
2
3  Class: Controller
4    EquivalentTo: controls some sys:Property
5    SubClassOf: produces some ControllerSignal
6
7  ObjectProperty: produces
8    Domain: Producer
9    Range: Signal
10
11 ObjectProperty: isProducedBy
12   InverseOf: produces
13
14 Class: Producer
15   EquivalentTo: produces min 1
16   SpinRule:
17     CONSTRUCT {
18       ?this :consumes ?negativeSignal
```

```
19      } WHERE {
20        ?this :produces ?signal .
21        ?negativeSignal expr:hasOperand ?signal .
22        ?negativeSignal expr:hasOperator math:unaryMinus
23      }
```

Listing 3.1: Small excerpt of the *control systems* ontology.

```
1  Ontology <http://www.mercator.iac.es/onto/metamodels/electricity>
2
3  Class: Insulator
4    DisjointWith: Conductor
5    SubClassOf: (isConnectedTo max 0) and (sys:hasPart only Insulator)
6
7  ObjectProperty: isConnectedTo
8    Characteristics: Symmetric
```

Listing 3.2: Small excerpt of the *electricity* ontology.

The above excerpts illustrate the following metaconcepts:

1. **Subsumption:** e.g. listing 3.1 line 5: a controller is-a thing that produces some signal.
2. **Equivalence:** e.g. listing 3.1 line 4: literally, this statement says that "the set of controllers is equivalent to the set of things that control some system property".
3. **Domains and ranges:** e.g. listing 3.1 lines 8-9: if `A produces B` then `A` must be an individual of the class `Producer` and `B` must be an individual of the class `Signal`.
4. **Cardinality:** e.g. listing 3.1 line 15: a producer produces minimum one "thing". Implicitly, this "thing" must be a `Signal`, because of the range of the `produces` property.
5. **Existential quantification:** e.g. listing 3.1 line 5: "`some`" implies that there exists at least one `ControllerSignal` that is produced by the `Controller`.
6. **Universal quantification:** e.g. listing 3.2 line 5: "`only`" implies that an `Insulator` is composed of nothing but other `Insulator`s.
7. **Symmetry:** e.g. listing 3.2 line 8: if `A isConnectedTo B`, then a reasoner will infer that `B isConnectedTo A`.
8. **Inversion:** e.g. listing 3.1 line 12: if `A produces B`, then a reasoner will infer that `B isProducedBy A`.

Several more "metaconcepts" can be represented by OWL; for the full list we refer to the OWL 2 Primer document [115]. One important metaconcept that is *not* represented by the built-in primitives of our selected knowledge

representation languages, is the *part-whole* relationship. As elaborated in [113], RDF Schema does provide the concept of membership (via the `rdfs:member` relationship and the `rdfs:Container` class), but this is arguably equivalent to part-whole composition since membership is not transitive (e.g. "the goose's leg is part of the goose but not part of the flock of geese" [113]). As will be seen in 3.3.1, our *systems* ontology therefore defines a part-whole relationship (called `hasPart`) using other metaconcepts such as asymmetry and transitivity.

To model constraints about the concepts of our domains of interest that could not be expressed using OWL and RDF Schema, we used SPIN: the SPARQL Inferencing Notation. SPIN allows the expression of if-then rules using SPARQL queries: see for instance listing 3.1 lines 16–23. This rule says that if an individual of the class `Producer` (assigned to the variable `?this`) *produces* a signal S, and there exists a negative signal $-S$, then this individual *consumes* $-S$. This rule is fired for all individuals of the class `Producer`. For every individual, the pattern of the `WHERE` clause is matched against the models. Every line of the `WHERE` clause is an expression consisting of known resources (such as `expr:hasOperand` and `math:unaryMinus`) and of variables (starting with a `?`). If the pattern matches, then the expressions of the `CONSTRUCT` clause are added to the models.

SPIN can be used to implement OWL 2 RL, which is a subset of the OWL 2 metaconcepts that can be represented by rules. SPIN uses the SPARQL query language to express rules. For instance, the SPARQL rule of listing 3.3 shows how *property ranges* can be represented: if `?p` has range `?c`, and `?x` is related to `?y` via `?p`, then `?y` is of type `?c`. SPIN provides the primitives to embed these queries within an ontology, by "binding" them to classes. For instance, the SPARQL rule of listing 3.3 can be bound to the `owl:Thing` class, thereby instructing the reasoner – a *rules engine* – that the rule must be executed for all individuals (since `owl:Thing` is the set of all possible individuals). In the same way, we can bind custom rules to our classes (as in listing 3.1, which displays a rule bound to the `Producer` class).

```
1  # rule "prp-rng"
2  CONSTRUCT {
3      ?y rdf:type ?c .
4  }
5  WHERE {
6      ?p rdfs:range ?c .
7      ?x ?p ?y .
8  }
```

Listing 3.3: SPARQL rule that represents *property ranges*.

SPIN also supports the concept of closed-world constraint verification. An example of such a rule is shown in listing 3.4. If the `WHERE` clause of the `SpinConstraint` rule of the `Requirement` class matches, then the `CONSTRUCT` clause will create a new individual `_:b0` that represents the constraint violation. Clearly, this rule assumes that all information is known and complete (i.e. it

assumes that the world is "closed") because otherwise it could never verify the `FILTER NOT EXISTS` statement. In an "open" world, it would be impossible to conclude that some pattern does *not* exist by "filtering" the known axioms! Often, constraint verification is very useful in the closed world, however. For instance, the rule of listing 3.4 would create a `ConstraintViolation` individual if a requirement (defined by our *development* ontology, see 3.3.13) is not satisfied, or if a requirement has a derived requirement that is not satisfied.

```
1  Class: Requirement
2    EquivalentTo: mod:represents some Constraint
3    SpinConstraint:
4      CONSTRUCT {
5        _:b0 a spin:ConstraintViolation .
6        _:b0 rdfs:label "Requirement is not satsified!"
7      } WHERE {
8        {
9          FILTER NOT EXISTS { ?this :isSatisfiedBy ?x }
10       }
11       UNION
12       {
13         FILTER NOT EXISTS { ?req :isDerivedFrom ?this .
14                             ?req :isSatisfiedBy ?y    }
15       }
16     }
```

Listing 3.4: Example of closed-world constraint verification.

3.2.2 Syntax

Roughly speaking, the languages that we selected can represent knowledge in three ways:

1. Frame-based representation, as displayed in listings 3.1 and 3.2, groups together information of a frame (such as `Class`, `ObjectProperty`, `DataProperty`, and `Individual`) via slots (such as `EquivalentTo`, `SubClassOf`, `DisjointWith`, ...). This grouping of information makes ontologies much easier to read and to understand compared to an ungrouped ("flat") list of axioms [46].

2. Description logic is used to represent the relations between the frames. For instance, listing 3.1 line 5 says that the set of `Insulator` individuals is a subset of the union of the set of individuals that are connected to a set that is known to be empty, and the set of individuals that consist of one or more insulators.

3. Rule-based representation is used to express knowledge in a very different way, via *if-then* (or rather: `CONSTRUCT {} WHERE {}`) rules. Because the language used to write these rules (SPARQL) is very expressive, but more verbose and less readable at the same time, we use it only to express knowledge that we cannot express using the other syntaxes.

Our choice for RDF-S, OWL 2 RL, and SPIN is thus not only based on semantics, but also on syntax. The versatility of the chosen knowledge representation languages allows us to structure our ontologies using frames, to express very readable relations between those frames using description logic, and to express some "special cases" – in both the open and the closed world – using rules.

All of the chosen knowledge representation languages are based on RDF, the Resource Description Framework. RDF is the specification of a data model: it specifies that statements about resources should be expressed as a "triple" consisting of a subject, predicate, and object. For instance, the definition of `Controller` of the *control systems* ontology (with prefix `ctrl`) as displayed before in listing 3.1 using frame-based Manchester syntax, can be represented by the triples of listing 3.5. The prefix descriptions of lines 1-5 can be omitted if the resources are written "in full" as URIs instead of QNames.

```
1  prefix ctrl:<http://www.mercator.iac.es/onto/metamodels/controlsystems#>
2  prefix sys :<http://www.mercator.iac.es/onto/metamodels/systems#>
3  prefix rdf :<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4  prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>
5  prefix owl :<http://www.w3.org/2002/07/owl#>
6
7  ctrl:Controller  rdf:type             owl:Class .
8  ctrl:Controller  rdfs:subClassOf      owl:Thing .
9  ctrl:Controller  rdfs:subClassOf      _:b0 .
10 ctrl:Controller  owl:equivalentClass  _:b1 .
11
12 _:b0  rdf:type            owl:Restriction .
13 _:b0  owl:onProperty      ctrl:produces .
14 _:b0  owl:someValuesFrom  ctrl:ControllerSignal .
15
16 _:b1  rdf:type            owl:Restriction> .
17 _:b1  owl:onProperty      ctrl:controls .
18 _:b1  owl:someValuesFrom  sys:Property .
```

Listing 3.5: Definition of `ctrl:Controller` as triples.

RDF can thus be serialized in multiple ways: as a "flat" list of triples, or as frames using Manchester syntax. Several more serialization formats exists however, such as RDF/XML, N3, Turtle, JSON-LD, and others. Some of these formats support "quads" instead of triples: they relate triples to their context, thereby forming "named graphs". The N-Quads format, for instance, stores knowledge as quads of the form: *subject predicate object graph*.

In our framework we used the Turtle format to serialize our metamodels, because it has a concise and readable notation. The models of the actual systems, on the other hand, are written using a set of DSLs via a custom developed library called Ontoscript[3]. As will be explained in section 3.4, when Ontoscript models are executed, they are converted into JSON-LD because the latter supports context, and because JSON-LD is an easy to generate and widely used serialization format. To quickly convert files from one syntax to another, we also created a small software program called rdfconvert[4].

[3] *Ontoscript* is open source and hosted at `https://github.com/IvS-KULeuven/ontoscript`.
[4] *Rdfconvert* is open source and hosted at `https://github.com/WimPessemier/rdfconvert`.

3.2.3 Tool support

An advantage of using standards is that they are used by a community, and therefore they are often well supported. Semantic Web standards such RDF, RDF Schema, OWL, SPARQL, etc. are very popular, and are therefore supported by a large amount of (open-source) software. Well-known tools are the Protégé ontology editor, reasoners such as Pellet, RACER, FaCT++, and complete "frameworks" that include RDF storage, reasoning and querying, such as Jena, GraphDB, Stardog, TopBraid Suite, RDF4J, and so on.

While we tested many of the tools named above during the development phase of our framework, we settled on the tools listed below when the framework was applied to the Mercator telescope. The tools were selected based on functional requirements (e.g. the reasoner must support OWL 2 RL) but also on many other "personal" criteria such as the perceived usability, familiarity with technologies, supportability (e.g. Python is very common at the Mercator telescope), and so on.

- TopBraid Composer was used to create and edit the metamodel ontologies. The Eclipse-based ontology editor has built-in support for expressing and reasoning OWL 2 RL axioms and SPIN rules, for executing SPARQL queries, and for serializing ontologies in our preferred formats Turtle and JSON-LD. TopBraid Composer is proprietary software and can be downloaded at `http://www.topquadrant.com`. The ontology editor has simplified the development of our ontologies, but it is not strictly needed since ontologies may also be developed using a simple text editor – especially if very "readable" syntaxes are used such as Manchester syntax, as shown by the dozens of code listings troughout this chapter.
- TopBraid SPIN API is the rules engine that was built into our OntoManager tool. SPIN API is open source software written in Java, it can be downloaded at `http://topbraid.org/spin/api/`.
- RDFLib is a Python libary that allows reading, writing, storing, and querying RDF data. It is heavily used by our OntoManager tool for various purposes, as will be elaborated in section 3.5. RDFLib is open source software and is hosted at `https://github.com/RDFLib`.

Of course, other tools might be available that also satisfy our functional requirements, and that offer some advantages (e.g. use less processing power, use less memory, are more maintainable, ...) compared to the ones that we selected for pragmatic reasons. As will be elaborated in chapter 6, there is room for improvement, and if a second version of our framework would ever be built, then several choices would have to be reassessed.

3.3 Metamodels: ontologies

In this section we will present the ontologies that we developed, and explain some of the rationale behind the many subjective choices that they embody. All code listings in this section are expressed using the frame-based Manchester syntax [114], but we added the slots `SpinRule` and `SpinConstraint` to be able to express SPIN rules and constraints more concisely. This "extended" Manchester syntax is only used informally, within this thesis. In reality, the ontologies are stored in Turtle format, which is less compact and less easy to read, but which is more expressive and better supported by the available tools. The full ontologies in Turtle format can be retrieved on-line, e.g. by pointing a web-browser to the ontology URIs. In this thesis however, we only display some excerpts of the ontologies, in Manchester syntax.

Table 3.1 lists the ontologies that we developed[5]. Many of these ontologies (although not all of them) reuse some of the concepts defined by other ontologies. The ontologies are roughly ordered from very abstract ontologies such as *systems* and *containers*, to very concrete ontologies such as *IEC 61131*.

Table 3.1: Overview of our developed ontologies.

Ontology	*Prefix*	*URI*[5]
Systems	`sys`	`~/systems`
Containers	`cont`	`~/containers`
Models	`mod`	`~/models`
Expressions	`expr`	`~/expressions`
Documents	`doc`	`~/documents`
Organizations	`org`	`~/organizations`
Finite state machines	`fsm`	`~/finitestatemachines`
Colors	`colors`	`~/colors`
Geometry	`geom`	`~/geometry`
Control systems	`ctrl`	`~/controlsystems`
Development	`dev`	`~/development`
Manufacturing	`man`	`~/manufacturing`
Mechanics	`mech`	`~/mechanics`
Electricity	`elec`	`~/electricity`
Software	`soft`	`~/software`
IEC 61131	`iec61131`	`~/iec61131`

A metamodel that is not listed in table 3.1 is the *Quantities, units, dimensions and data types (QUDT)* ontology. This ontology is described in 3.3.4 since it is very important to our framework, but we did not add it to the table since we did not create it ourselves. Another metamodel that is *not* described in this section, is the underlying language of our DSLs: CoffeeScript. As will be

[5]For compactness, the URIs in this table are shortened by the ~ symbol, which stands for `http://www.mercator.iac.es/onto/metamodels`

elaborated in section 3.4, we frequently use the built-in primitives and the syntax of CoffeeScript when modeling our actual systems, e.g. to build "factories" that create models based on some input parameters.

3.3.1 Systems

In 3.1.1 we argued that "everything is a system", so a *systems* ontology is logically one of the most abstract or "top-level" ontologies to start with. Its purpose is to allow us to represent a system according to the definition given in 3.1.1, saying that "a system is something that is composed of elements which are directly or indirectly related to each other." We defined the property `hasElement` (to represent the *is composed of* relationship) and the classes `System` and `Element`: see listing 3.6. We constrained the `hasElement` relationship by saying it is asymmetric: nothing is composed of its composition. We defined *parts* and *properties* as specialized elements. Transitivity and irreflexivity are characteristics that differentiate properties and parts of the system: `hasPart` is a transitive subproperty of `hasElement`, while `hasProperty` is an irreflexive subproperty of `hasElement` that is disjoint with `hasPart`. For example, the primary mirror is a *part* of the optics of the telescope, and the optics is a *part* of the telescope, therefore the mirror is a *part* of the telescope. Conversely, the weight of the optics is a *property* of the optics, but it is obviously not a *property* of the telescope, nor a *property* of itself.

```
1  Ontology <http://www.mercator.iac.es/onto/metamodels/systems>
2    Import: <http://spinrdf.org/spin>
3    Import: <http://spinrdf.org/spin/owlrl-all>
4
5  Class: System
6    EquivalentTo: hasElement some Element
7
8  ObjectProperty: hasElement
9    Characteristics: Asymmetric
10   Domain: System
11   Range: Element
12
13 Class: Element
14   EquivalentTo: owl:Thing
15
16 Class: Part
17   EquivalentTo: isPartOf some System
18
19 ObjectProperty: hasPart
20   SubPropertyOf: hasElement
21   Characteristics: Asymmetric, Transitive
22   Domain: System
23   Range: Part
24
25 ObjectProperty: isPartOf
26   InverseOf: hasPart
27
28 Class: Property
29   EquivalentTo: isPropertyOf some System
30
31 ObjectProperty: hasProperty
32   Characteristics: Asymmetric, Irreflexive
33   SubPropertyOf: hasElement
```

```
34    DisjointWith: hasPart
35
36  ObjectProperty: isPropertyOf
37    InverseOf: hasProperty
```

Listing 3.6: The systems ontology (sys).

Realization

Another represented relationship is what we call `realize`, similar to what is often called "instanceOf" or "conformsTo". If system A realizes system B, it means that all parts and properties of B must be realized by an element of A. Realization is a strong relationship because it operates recursively, as also every part and property of the realized elements of B must be realized, and so on. Realization is reused for example:

- by the *development* ontology, to express that a design which realizes a concept must realize every element (such as its requirements, states, constraints, ...) of that concept;
- by the *software* ontology, to express that an instance of a software class must realize every attribute of that class;
- whenever an "instance" of something must be represented, such as a particular instance of an I/O module of a certain type.

A realization strongly ties a system to another system, saying that all parts and properties of the *realized* system must be traceable to an element of the *realizing* system. The latter is thus always equal or larger in size (i.e. it contains equal or more elements) than the former. Realization "by itself" does not mean a lot, but it can easily be reused by other ontologies, as illustrated by the examples above. In 5.1.2 of the evaluation chapter we will present an alternative definition that constrains the semantics of a realization much more.

The *systems* ontology defines a SPIN constraint to verify if all parts and properties of the realized system are indeed realized by the elements of the realizing system: see listing 3.7. We introduced the class `sys:Complete` to describe a system that can be considered as "completely" modeled: i.e. for which the world is assumed to be "closed". If this is the case, then the `SpinConstraint` rule will produce a new individual (`_:b0`) of the `ConstraintViolation` class, and add it to the knowledge base, whenever any property or part of the realized system is not realized by an element of the realizing system.

```
38  Class: Realization
39    EquivalentTo: realizes some System
40    SpinConstraint:
41      CONSTRUCT {
42        _:b0 a spin:ConstraintViolation .
```

```
        _:b0 spin:violationRoot ?this .
        _:b0 spin:violationPath sys:hasElement .
        _:b0 spin:violationValue ?e .
        _:b0 spin:violationLevel spin:Error .
        _:b0 rdfs:label ?msg
      }
      WHERE {
        ?this rdf:type :Complete .
        ?this :realizes ?other .
        ?other :hasPart|:hasProperty ?e .
        FILTER NOT EXISTS { ?this :hasElement*/:realizes ?e } .
        BIND (fn:concat("Realization error: no element which realizes ", ?e,
                        " (which is an element of ", ?other, " )")
              AS ?msg)
      }

Class: Complete

ObjectProperty: realizes
  Characteristics: Asymmetric
  Domain: Element
  Range: Element

ObjectProperty: isRealizedBy
  InverseOf: realizes
```

Listing 3.7: The systems ontology (sys): realization.

Interfaces

Finally, the systems ontology also defines the concept of an *interface*, see listing 3.8. An interface is a system that represents the boundary between two systems. Thus, every element of the interface must be a common element of both interfaced systems. We can implement this in two ways. One option is to enforce it: when we loop trough all elements of the interface, we can add an `isElementOf` relation from those elements to each interfaced system. Another option is to verify the interface (in the closed world!): by checking whether or not all interface elements are already related to both interfaced systems via `isElementOf` relations. We chose the second option, since the `isElementOf` property has very weak semantics, and therefore it is mostly used as a super-property of properties with stronger semantics. Verifying such a property is much more useful than enforcing it.

```
Class: Interface
  SubClassOf: System
  EquivalentTo: interfaces min 2
  SpinRule:
    CONSTRUCT {
      ?sys1 isInterfacedWith ?sys2
    } WHERE {
      ?this interfaces ?sys1 .
      ?this interfaces ?sys2 .
      FILTER(?sys1 != ?sys2)
    }
  SpinConstraint:
    CONSTRUCT {
      _:b0 a spin:ConstraintViolation .
      _:b0 rdfs:label "Element is no element of interfaced system!"
```

```
16      } WHERE {
17        ?this :interfaces ?system .
18        ?this :hasElement ?element .
19        FILTER NOT EXISTS { ?system :hasElement ?element }
20      }
21
22  ObjectProperty: interfaces
23    Domain: Interface
24    Range: System
25
26  ObjectProperty: isInterfacedWith
27    Characteristics: Symmetric
28    Domain: System
29    Range: System
```

Listing 3.8: The systems ontology (sys): interfaces.

3.3.2 Containers

The purpose of the *containers* ontology is to represent containers, items, and the containment relations between them. Containment is different from the part-whole relationship (`hasPart`) of the systems ontology. For instance, a room may contain a person, but the person is not a part of this room. The containment relationship is also different from the `hasProperty` relationship of the systems ontology, because containment it is transitive (i.e. if a building contains a room, and the room contains a person, then the building contains this person) and irreflexive (i.e. a room doesn't contain itself). We further constrain containment by saying that if an item is contained by two containers, then these containers must "fully overlap": one of them must contain the other one. This differentiates the `contains` relationship from the `hasPart` relationship of the *systems* ontology. To be able to express this efficiently, we created an `containsOrIsContainedBy` relationship, which serves as the super-property of both `contains` and `isContainedBy`. In the closed world, we can generate a warning if two containers are related by `containsOrIsContainedBy` and it is not clear which container contains the other one. Listing 3.9 shows the resulting *containers* ontology.

```
1   Ontology <http://www.mercator.iac.es/onto/metamodels/containers>
2     Import: <http://www.mercator.iac.es/onto/metamodels/systems>
3
4   Class: Container
5     EquivalentTo: contains some Item
6     SpinConstraint:
7       CONSTRUCT {
8         _:b0 a spin:ConstraintViolation .
9         _:b0 rdfs:label "Unspecified containsOrIsContainedBy relation"
10      } WHERE {
11        ?this :containsOrIsContainedBy ?other .
12        FILTER NOT EXISTS { ?this :contains|:isContainedBy ?other }
13      }
14
15  Class: Item
16    EquivalentTo: isContainedBy some Container
17    SpinRule:
18      CONSTRUCT {
```

```
      ?c1 :containsOrIsContainedBy ?c2 .
    } WHERE {
      ?this :isContainedBy ?c1 .
      ?this :isContainedBy ?c2 .
      FILTER(?c1 != ?c2)
    }

ObjectProperty: contains
  SubPropertyOf: containsOrIsContainedBy, hasElement
  Characteristics: Transitive, Irreflexive
  Domain: Container
  Range: Item
  SubPropertyChain: hasContainment o hasItem

ObjectProperty: isContainedBy
  SubPropertyOf: containsOrIsContainedBy, isElementOf
  InverseOf: contains

ObjectProperty: containsOrIsContainedBy
  SubPropertyOf: hasElement
  Characteristics: Symmetric
```

Listing 3.9: The containers ontology (cont).

Containment is an n-ary relationship: it relates to more than just the subject and object of a binary relation. For instance, a container may be ordered, which implies that the containment relations defined by the container relate to some data that allows ordering. If this data are numbers, then we call the ordered container a `List`: see listing 3.10. It's important to realize that these numbers characterize the *containment*, and not the *item* itself. This can easily be verified by considering an item that is contained by two fully overlapping lists: if the ordering numbers would be linked to the items instead of the containment relations, then it would be impossible to know which number belongs to which list. We therefore created a special class (`Containment`) to represent the containment relationship explicitly. For each item of a container, there is a `Containment` individual which may be related to "ordering" data such as a number. Containment individuals are related to exactly one container and exactly one item via so-called *inverse functional properties* – properties which cannot relate two distinct subjects to the same object.

```
Class: Containment
  EquivalentTo: isContainmentOf exactly 1
                and hasItem exactly 1
  DisjointWith: Container, Item

ObjectProperty: hasContainment
  Characteristics: InverseFunctional
  Domain: Container
  Range: Containment

ObjectProperty: hasItem
  Characteristics: InverseFunctional
  Domain: Containment
  Range: Item

ObjectProperty: isContainmentOf
  InverseOf: hasContainment

ObjectProperty: isItemOf
```

```
    InverseOf: hasItem

  Class: List
    EquivalentTo: hasContainment only (hasNumber exactly 1)

  DataProperty: hasNumber
    Domain: Containment
    Range: xsd:integer
```

Listing 3.10: The containers ontology (cont): containment.

3.3.3 Models

We created a *models* ontology based on the informal definitions of 3.1.1. Its purpose is to provide a basic vocabulary that can be reused by less abstract ontologies that depend on the notion of models. For instance, our *organizations* ontology defines a *role* as a model (i.e. something that represents a person for a specific purpose), and our *development* ontology defines a *requirement* as something that represents a constraint of a system. We could only little constrain the `represents` relationship and the `Purpose` class, resulting in poor formal semantics.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/models>
  Import: <http://www.mercator.iac.es/onto/metamodels/systems>

Class: Model
  EquivalentTo: represents some sys:System
                and hasPurpose some Purpose

Class: Purpose
  EquivalentTo: isPurposeOf min 1

ObjectProperty: represents
  Characteristics: Asymmetric, Irreflexive, Transitive

ObjectProperty: hasPurpose
  SubPropertyOf: hasProperty
  Range: Purpose

ObjectProperty: isPurposeOf
  InverseOf: hasPurpose
```

Listing 3.11: The models ontology (mod).

3.3.4 Quantities, units, dimensions and data types

The unambiguous representation of quantities, units, dimensions and data types is very much needed in complex engineering projects, as shown by infamous events such as the loss of NASA's Mars Climate Orbiter in 1999 due to inconsistent use of metric and SI units in its ground-based software [100]. Fortunately, a suitable and very elaborate set of ontologies is already

publicly available: the *quantities, units, dimensions and data types* ontologies (or *QUDT*), developed by the TopQuadrant company and NASA.

Below we list the most important concepts of the QUDT ontologies, as described in [45]:

- `QuantityKind`: quantity kinds are observable properties that can be measured and quantified numerically, such as length, mass, time, force, energy, and so on;
- `Quantity`: a quantity is the measurement of an observable property of a particular object, event, or physical system, such as the mechanical power of a specific motor of a telescope;
- `QuantityValue`: a quantity value represents the value of a quantity, having a numeric value and a unit;
- `QuantityDimensionVector`: a dimension vector allows the definition of a quantity as a product of the 7 basic physical dimensions (e.g. the dimension of *force* is $mass \times acceleration = mass^1 \times length^1 \times time^{-2}$, resulting in a vector $[1, 1, -2, 0, 0, 0, 0]$);
- `Unit`: over 1400 units are defined by QUDT, and include information such as conversion multipliers and offsets (e.g. the `DegreeFahrenheit` unit has a conversion multiplier of 0.555... and a conversion offset of 255.370370... because $k = 255.370370... + f \times 0.555...$, with f a value in degrees Fahrenheit and k a value in Kelvin).

A single quantity may have multiple quantity values: for instance, the temperature of a detector of an instrument may have a quantity value in Kelvin and a quantity value in degrees Celsius. A unit conversion ontology (`http://qudt.org/spin/unitconversion`) is even available to convert such quantity values from one unit to another by the reasoning engine, using SPIN rules. Because our framework is based on SPIN, we are able to reuse those features very easily.

3.3.5 Expressions

The purpose of our *expressions* ontology is to provide a vocabulary for expressing *boolean expressions* (i.e. propositional formulas, having a truth value), *operations* (e.g. unary and binary operations, having an operator and operands), and specific operations such as $\vee, \wedge, \neg, \rightarrow, \leftrightarrow, =, <, \leq, >, \geq, :=, \mathcal{U}, \Diamond$, and $\Box$. All expressions are (explicit or implicit) subclasses of the `Expression` class, which is restricted by the rule saying that – at the time of reasoning – expressions can only have one value. A `Primitive` is a kind of expression that is "atomic", i.e. it cannot be broken down into more expressions. See listing 3.12 for their formal definitions.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/expressions>
  Import: <http://www.mercator.iac.es/onto/metamodels/systems>

Class: Expression
  EquivalentTo: Primitive or sys:hasPart only Expression
  SpinConstraint:
    CONSTRUCT {
      _:b0 a spin:ConstraintViolation .
      _:b0 rdfs:label "Multiple (different) values!"
    } WHERE {
      ?this expr:hasValue ?v0 .
      ?this expr:hasValue ?v1 .
      FILTER (?v0 != ?v1)
    }

Class: Primitive
  EquivalentTo: Expression and (sys:hasPart max 0 expr:Expression)

DataProperty: hasValue
  Domain: Expression

DataProperty: hasNumericValue
  Domain: Expression
  SubPropertyOf: hasValue
  EquivalentTo: qudt:numericValue
```

Listing 3.12: The expressions ontology (expr).

Boolean expressions

Another kind of expressions are the boolean expressions: they can only be *true* or *false* (as defined by XSD, the XML Schema specifications [112]) at the time of reasoning. See listing 3.13.

```
Class: BoolExpression
  EquivalentTo: hasValue some xsd:boolean

Class: True
  SubClassOf: BoolExpression
  EquivalentTo: hasValue value "true"^^xsd:boolean
  DisjointWith: False

Class: False
  SubClassOf: BoolExpression
  EquivalentTo: hasValue value "false"^^xsd:boolean
  DisjointWith: True
```

Listing 3.13: The expressions ontology (expr): boolean expressions.

Operations

To express operations, we say that an `Operation` consists of at least one `Operand` and exactly one `Operator`. We also defined *unary operations* (having only one operand) and *binary operations* (having two operands). For unary and binary operations, we reuse the operator (an `owl:Thing`) as a relationship

(an `owl:ObjectProperty`) between the operation and operand (for unary operations) and between both operands (for binary operations). We added SPIN rules to infer these (redundant) relations, because they simplify pattern matching when writing queries or additional SPIN rules.

```
38  Class: Operation
39    EquivalentTo: (hasOperand some Operand)
40                  and (hasOperator exactly 1 Operator)
41
42  Class: Operand
43    SubClassOf: Expression
44    SubClassOf: isOperandOf some Expression
45
46  Class: Operator
47    SubClassOf: isOperatorOf some Expression
48
49  Class: UnaryOperation
50    SubClassOf: hasOperator exactly 1 UnaryOperator
51    EquivalentTo: hasOperand exactly 1
52    SpinRule:
53      CONSTRUCT {
54        ?this ?operator ?operand .
55      } WHERE {
56        ?this :hasOperand ?operand .
57        ?this :hasOperator ?operator
58      }
59
60  Class: BinaryOperation
61    SubClassOf: hasOperator exactly 1 BinaryOperator
62    EquivalentTo: hasOperand exactly 2
63    SpinRule:
64      CONSTRUCT {
65        ?left ?operator ?right .
66      } WHERE {
67        ?this :hasLeftOperand ?left .
68        ?this :hasRightOperand ?right .
69        ?this :hasOperator ?operator
70      }
71
72  Class: UnaryOperator
73    EquivalentTo: isOperatorOf some UnaryOperation
74
75  Class: BinaryOperator
76    EquivalentTo: isOperatorOf some BinaryOperation
77
78  ObjectProperty: hasOperand
79    SubPropertyOf: sys:hasElement
80    Domain: Operation
81    Range: Operand
82
83  ObjectProperty: hasLeftOperand
84    SubPropertyOf: sys:hasOperand
85    Domain: BinaryOperation
86
87  ObjectProperty: hasRightOperand
88    SubPropertyOf: sys:hasOperand
89    Domain: BinaryOperation
90    DisjointWith: hasLeftOperand
```

Listing 3.14: The expressions ontology (expr): operations.

Logic operations and assignment

Our *expressions* ontology defines the logic operations `And`, `Or`, `Not`, `Implication`, and `Equivalence`. We also defined `Assignment`, which is more than just a logic operation as it "copies" any kind of value of its right operand to its left operand. In our ontology, we implemented several inference rules that constitute the semantics of the operations. As illustrated in 3.15, most of the rules are implemented using SPIN. For some rules, however, even OWL is sufficiently expressive. For instance, the `isEquivalentTo` operator is modeled as a symmetric subproperty of the `implies` operator, and therefore if `A isEquivalentTo B` then it follows that `A implies B` and `B implies A`. The `isAssignedTo` operation on the other hand is transitive, since if `A isAssignedTo B` and `B isAssignedTo C` then `A isAssignedTo C`.

Due to the large amount of SPIN rules, the ontology definitions are very elaborate, and therefore we only show an excerpt of the ontology below (see listing 3.15). The full ontology can be accessed online via its URI (`http://www.mercator.iac.es/onto/metamodels/expressions`). A summary of the operations, the operators, their semantics, and the inference rules that we implemented using SPIN rules or using RDF Schema and OWL, is shown in table 3.2.

```
91  Class: And
92    SubClassOf: BinaryOperation
93    EquivalentTo: hasOperator value and
94    SpinRule:
95      # Conjunction elimination
96      CONSTRUCT {
97        ?left :hasValue true .
98        ?right :hasValue true .
99      } WHERE {
100       ?this :hasValue true .
101       ?this :hasLeftOperand ?left .
102       ?this :hasRightOperand ?right
103     }
104   SpinRule:
105     # Conjunction introduction
106     CONSTRUCT {
107       ?this :hasValue true .
108     }
109     WHERE {
110       ?this :hasLeftOperand/:hasValue true .
111       ?this :hasRightOperand/:hasValue true
112     }
113   # and so on ...
114
115 Individual: and
116   Types: BinaryOperator, ObjectProperty
117
118 # ... similar for Or, Not, Implication, Equivalence
```

Listing 3.15: The expressions ontology (expr): logic operations and assignment.

Table 3.2: Summary of the implemented logic operations and assignment

`And`	*Operator:*	`and`
	Semantics:	$\wedge$, logical conjunction
	Rules:	$(P \wedge Q) \vdash P$ and $(P \wedge Q) \vdash Q$
		$P, Q \vdash P \wedge Q$
		$(\neg P) \vdash \neg(P \wedge Q)$ and $(\neg Q) \vdash \neg(P \wedge Q)$
		$(P \wedge Q), (Q \rightarrow \neg R) \rightarrow (P \wedge Q) \rightarrow \neg R$
`Or`	*Operator:*	`or`
	Semantics:	$\vee$, logical disjunction
	Rules:	$(P \rightarrow Q), (R \rightarrow Q), (P \vee R) \vdash Q$
		$P \vdash P \vee Q$ and $Q \vdash P \vee Q$
		$(\neg P) \vdash \neg(P \vee Q)$
		$\neg(P \vee Q) \vdash (\neg P), (\neg Q)$
`Not`	*Operator:*	`not`
	Semantics:	$\neg$, negation
	Rules:	$\neg\neg P \vdash P$
`Implication`	*Operator:*	`implies`
	Semantics:	$\rightarrow$, material implication, if then
	Rules:	$(P \rightarrow Q), P \vdash Q$
`Equivalence`	*Operator:*	`isEquivalentTo`
	Semantics:	$\leftrightarrow$, material equivalence, if and only if
	Rules:	$(P \leftrightarrow Q) \vdash (P \rightarrow Q), (Q \rightarrow P)$
		$(P \rightarrow Q), (Q \rightarrow P) \vdash (P \leftrightarrow Q)$
`Assignment`	*Operator:*	`isAssignedTo`
	Semantics:	$:=$, assignment, valuation
	Rules:	$(P := Q), (Q := R) \vdash (P := R)$
		$(P := Q), (Q \rightarrow R) \vdash (P \rightarrow R)$
		$(P \rightarrow Q), (Q := R) \vdash (P \rightarrow R)$

Comparison operations

Our ontology defines five binary comparison operations: `Equality`, `GreaterThan`, `GreaterThanOrEqualTo`, `LessThan`, and `LessThanOrEqualTo`. For each operation, we implemented rules to infer the comparison relation between two expressions, based on the subject of the `hasValue` property (see listing 3.12 for the definition of `hasValue`). We also implemented rules that create constraint violations, if two expressions are said to be equal (or greater than, or smaller than, etc.) while their values reveal the opposite. Since the rules are implemented using SPIN, the exact semantics of each operation (i.e. $=$, $>$, $\geq$, $<$, $\leq$) is thus defined by the SPARQL specifications [118]. In listing 3.16 we only show the definition of `Equality`. The other comparison operations are very similar, and can be accessed online via the URI of the ontology.

```
Class: Equality
  SubClassOf: BinaryOperation
  EquivalentTo: hasOperator value isEqualTo
  SpinRule:
    # both operands of a true equality have the same values
    CONSTRUCT {
      ?left expr:hasValue ?rightValue .
      ?right expr:hasValue ?leftValue .
    } WHERE {
      ?this expr:hasValue true .
      ?this expr:hasLeftOperand ?left .
      ?left expr:hasValue ?leftValue .
      ?this expr:hasRightOperand ?right .
      ?right expr:hasValue ?rightValue
    }
  SpinRule:
    # if both operand values are equal, then the equality is true
    CONSTRUCT {
      ?this expr:hasValue true .
    } WHERE {
      ?this expr:hasLeftOperand/expr:hasValue ?leftValue .
      ?this expr:hasRightOperand/expr:hasValue ?rightValue .
      FILTER (?leftValue = ?rightValue)
    }
  SpinConstraint:
    # both operands of a true equality must have the same values
    CONSTRUCT {
      _:b0 a spin:ConstraintViolation .
      _:b0 rdfs:label "Both operands must have the same values!"
    } WHERE {
      ?this expr:hasValue true .
      ?this expr:hasLeftOperand/expr:hasValue ?leftValue .
      ?this expr:hasRightOperand/expr:hasValue ?rightValue .
      FILTER (?leftValue != ?rightValue)
    }

# ... similar for GreaterThan, GreaterThanOrEqualTo, ...
```

Listing 3.16: The expressions ontology (expr): comparison operations.

Temporal logic operations

We investigated the use of metric temporal Logic (MTL) to specify the time-dependent constraints of real-time systems. MTL is a real-time extension of linear temporal logic (LTL), a formalism which considers time as a series of clock ticks to allow specification and verification of "clocked" systems. MTL on the other hand models the run of a system either as a sequence of events that are time-stamped with reals (so-called *pointwise semantics*) or as a trajectory with domain the set $\mathbb{R}_+$ of non-negative reals (so-called *continuous semantics*) [79]. Of the many dialects of MTL, we selected signal temporal logic (STL) as defined by Maler and Nickovic [62] to be represented by our ontology. STL restricts the temporal modalities to intervals of the form $[a, b]$ with $0 \leq a < b$ and $a, b \in \mathbb{Q}_{\geq 0}$, and thereby avoids some of the problems of the unbounded modalities of the standard semantics for temporal logic (see [62]). Our ontology represents three STL temporal operations (see also listing 3.17):

- $\varphi \mathcal{U}_{[a,b]} \psi$: the binary $\text{Until}_{[a,b]}(\varphi, \psi)$ operation says that there exists a time t' of the interval $[a, b]$ at which ψ holds, and that φ holds continuously in

the interval $[a, t']$[6]. In other words: φ has to hold *until* ψ holds, and at that time they both hold.

- $\Diamond_{[a,b]}\varphi$: the unary Eventually$_{[a,b]}(\varphi)$ operation says that there exists a time t' of the interval $[a, b]$ at which φ holds. In other words: signal φ must *eventually* hold within the interval. This operation can easily be derived from the *until* operation: $\Diamond_{[a,b]}\varphi = \top \mathcal{U}_{[a,b]}\varphi$ (where $\top$ stands for the boolean *true* value).
- $\Box_{[a,b]}\varphi$: the unary Always$_{[a,b]}(\varphi)$ operation says that φ must hold for every time t' of the interval $[a, b]$. This operation can be derived from the *eventually* operation: $\Box_{[a,b]}\varphi = \neg\Diamond_{[a,b]}\neg\varphi$, where $\neg$ stands for the unary *not* operation.

More formal definitions of STL and the above operations can be found in [62]. In our framework, we use these operations to model various constraints of our systems. For instance, the following expression:

$$\Box((startOpening \wedge enabled) \rightarrow \Diamond_{[0,60s]}(open \vee error \vee aborted))$$

says that the *open* or *error* or *aborted* state of a system must always become active, at the latest 60 seconds after the *startOpening* and *enabled* states have been active momentarily. Despite the apparent simplicity of the operators, specification and verification of these constraints remains a very difficult task: aside from the complexity of the logic, there is the question of how accurately our models represent the actual systems (which have to deal with inconsistency of time throughout the distributed control system, the finite sampling of time by the components of the system, and so on). For instance, in the above example it is not specified how long ($startOpening \wedge enabled$) should be active to be detectable by the system, or how long ($open \vee error \vee aborted$) has to be active for the system to respond. Additional constraints such as

$$\Box(open \rightarrow \Box_{[0,0.01s]} openSignal)$$

may specify a minimum time for signals (such as boolean software variables) to be active, but they increase the complexity of the constraints specification of a system significantly.

```
156  Class: Until
157    SubClassOf: BinaryOperation and hasInterval max 1
158    EquivalentTo: hasOperator value until
159
160  Class: Eventually
161    SubClassOf: UnaryOperation and hasInterval max 1
162    EquivalentTo: hasOperator value eventually
163
164  Class: Always
165    SubClassOf: UnaryOperation and hasInterval max 1
166    EquivalentTo: hasOperator value always
```

[6] As noted by the creators of STL, unlike the conventional definition of *until*, there exists a time at which both φ and ψ hold.

```
167
168  ObjectProperty: hasInterval
169    Characteristics: Functional
170    Range: Interval
171
172  Class: Interval
173    EquivalentTo: (hasLeftBound exactly 1)
174                  and (hasRightBound exactly 1)
175
176  DataProperty: hasLeftBound
177    Characteristics: Functional
178    Domain: Interval
179
180  DataProperty: hasRightBound
181    Characteristics: Functional
182    Domain: Interval
```

Listing 3.17: The expressions ontology (expr): temporal logic operations.

Data types

Our expressions ontology finally also defines a set of data types, and a corresponding set of `Primitive` subclasses that represent primitives with a specific data type. The data types are based on XSD, the XML Schema specifications [112]. For each data type, we implemented an inference rule using SPIN, to classify a primitive based on the data type of the subject of the `hasValue` property. Ontologies such as those that represent programming languages, can easily relate their specific data types to the "shared" data types of the *expressions* ontology. For instance, the IEC 61131-3 programming language specifications have a data type called `UDINT`, which we related to the `t_uint32` individual of the expressions ontology via the `owl:sameAs` relation. A summary of the data types of the *expressions* ontology is shown in table 3.3.

Table 3.3: Data types of the *expressions* ontology

Class	*Individual*	*Data type of the* `hasValue` *subject*
`Bool`	`t_bool`	xsd:boolean
`UInt8`	`t_uint8`	xsd:unsignedByte
`Int8`	`t_int8`	xsd:byte
`UInt16`	`t_uint16`	xsd:unsignedShort
`Int16`	`t_int16`	xsd:short
`UInt32`	`t_uint32`	xsd:unsignedInt
`Int32`	`t_int32`	xsd:int
`UInt64`	`t_uint64`	xsd:unsignedLong
`Int64`	`t_int64`	xsd:long
`Float`	`t_float`	xsd:float
`Double`	`t_double`	xsd:double
`String`	`t_string`	xsd:string
`ByteString`	`t_bytestring`	xsd:hexBinary

3.3.6 Mathematics

The purpose of the *mathematics* ontology is to allow us to express mathematical formulas. It extends the *expressions* ontology, because mathematical operations are defined as subclasses of `expr:UnaryOperation` or `expr:BinaryOperation`, with a mathematical `Operator` such as `plus`, `minus`, `times`, `dividedBy`, and so on. Table 3.4 summarizes the operations that we defined. It also lists the rules that we implemented for each operation using SPIN: if x, y, z are `qudt:Quantity` or `qudt:QuantityValue` individuals with known numeric values, then the numeric value of $\bullet$ can be inferred by the reasoner. The table shows that only the most basic inference rules have been implemented, because SPARQL (the underlying language of SPIN) is not sufficiently expressive to valuate operations such as *sine* and *square root*. In principle however, it is possible to implement these operations by using the much more expressive javascript extensions of SPIN. We refrained from doing this however, since the added value of having these operations valuated by the inference engine is, with our application in mind, very limited.

Table 3.4: Operations and operators of the *mathematics* ontology

Operation	*Operator*	*Semantics*	*Rules implemented*
`Addition`	`plus`	$x + y$	$x + y = \bullet$
			$x + \bullet = z$
			$\bullet + y = z$
`Subtraction`	`minus`	$x - y$	$x - y = \bullet$
			$x - \bullet = z$
			$\bullet - y = z$
`Multiplication`	`times`	$x \times y$	$x \times y = \bullet$
			$x \times \bullet = z$
			$\bullet \times y = z$
`Division`	`dividedBy`	x/y	$x/y = \bullet$
			$x/\bullet = z$
			$\bullet/y = z$
`UnaryMinus`	`unaryMinus`	$-x$	$-x = \bullet$
`Abs`	`absOf`	$\|x\|$	$\|x\| = \bullet$
`Power`	`exponent`	x^y	
`Square`	`squareOf`	x^2	
`SquareRoot`	`squareRootOf`	$\sqrt{x}$	
`Sine`	`sineOf`	$sin(x)$	
`Cosine`	`cosineOf`	$cos(x)$	
`Tangent`	`tangentOf`	$tan(x)$	
`Cotangent`	`cotangentOf`	$cot(x)$	
`ArcSine`	`arcSineOf`	$arcsin(x)$	
`ArcCosine`	`arcCosineOf`	$arccos(x)$	
`ArcTangent`	`arcTangentOf`	$arctan(x)$	
`ArcCotangent`	`arcCotangentOf`	$arccot(x)$	

3.3.7 Documents

To be able to model documents such as data sheets, manuals, schematics, etc., we created a *documents* ontology. It defines a class called `Document`, by stating that it is equivalent to the `Document` class of the *friend of a friend (FOAF)* ontology[7]. FOAF is a widely used ontology to describe persons, groups of persons, and some objects such as documents. Aside from the `Document` class, we also defined two data properties for specifying the name of a document, and the filename of a digital document.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/documents>
  Import: <http://www.mercator.iac.es/onto/metamodels/models>
  Import: <http://xmlns.com/foaf/0.1/>

Class: Document
  EquivalentTo: foaf:Document
  SubClassOf: mod:Model

DataProperty: hasName
  Domain: Document
  Range: xsd:string

DataProperty: hasFileName
  Domain: Document
  Range: xsd:string
```

Listing 3.18: The documents ontology (doc).

3.3.8 Organizations

To represent organizations, persons, and roles of persons, we created an *organizations* ontology. According to the ontology, these concepts are so-called *named entities*: they have at least one name, and optionally a long name and/or short name. The only formal constraint is that long names cannot be shorter than short names. We defined `Person` by saying it is equivalent to the concept of *person* according to FOAF, the *friend of a friend* ontology. An `Organization` is then simply something that *contains* at least one person (and therefore it is a `cont:Container`), and a `Role` is something that *represents* a person. See listing 3.19.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/organizations>
  Import: <http://xmlns.com/foaf/0.1/>
  Import: <http://www.mercator.iac.es/onto/metamodels/models>
  Import: <http://www.mercator.iac.es/onto/metamodels/containers>

Class: NamedEntity
  EquivalentTo: hasName min 1
  SpinConstraint:
    CONSTRUCT {
      _:b0 a spin:ConstraintViolation .
```

[7]FOAF URI: `http://xmlns.com/foaf/0.1/`, website: `http://www.foaf-project.org`.

```
        _:b0 rdfs:label "Long name shorter than short name!"
      } WHERE {
        ?this :hasLongName ?long .
        ?this :hasShortName ?short .
        FILTER( strlen(?long) < strlen(?short) )
      }

DataProperty: hasName
  SubPropertyOf: rdfs:label
  EquivalentTo: foaf:name
  Range: xsd:string

DataProperty: hasLongName
  SubPropertyOf: hasName

DataProperty: hasShortName
  SubPropertyOf: hasName

Class: Role
  SubClassOf: NamedEntity
  EquivalentTo: (mod:represents some Person)
                and (mod:hasPurpose some mod:Purpose)

Class: Person
  SubClassOf: NamedEntity
  EquivalentTo: foaf:Person

Class: Organization
  SubClassOf: NamedEntity
  SubClassOf: cont:contains some Person
```

Listing 3.19: The organizations ontology (org).

3.3.9 Finite state machines

The purpose of our *finite state machines* ontology is to allow us to describe the behavior of a system in a very simple and convenient way, using states and transitions. According to our ontology, a `StateMachine` consists of a number of `State` individuals, which represent boolean expressions of properties (i.e. `sys:Property` individuals) of the machine. The state of a system is thus represented by the combination of the truth values of all `State` individuals of the system. If a system is considered as a state machine, then its non-boolean properties are thus "abstracted away". For instance, the state of a simple state machine model of a motor may be represented as *unlocked* and *moving* and *temperatureOK*. The measured velocity and temperature of the motor (perhaps represented by integers consisting of 2^{16} possible "states") are thus not considered anymore. Our ontology further defines that a state is called an `ActiveState` when it is *true* at reasoning time. A `SuperState` on the other hand is a state that is also a state machine (e.g. a state of a motor called *moving* may be a state machine of the states *movingForward* and *movingBackwards*).

```
Ontology <http://www.mercator.iac.es/onto/metamodels/finitestatemachines>
  Import: <http://www.mercator.iac.es/onto/metamodels/models>
  Import: <http://www.mercator.iac.es/onto/metamodels/expressions>

Class: StateMachine
```

```
    EquivalentTo: hasState some State

  ObjectProperty: hasState
    SubPropertyOf: sys:hasProperty
    Domain: StateMachine
    Range: State

  ObjectProperty: isStateOf
    InverseOf: hasState

  Class: State
    EquivalentTo: expr:BoolExpression and isStateOf some StateMachine

  Class: StateVariable
    SubClassOf: sys:isPropertyOf some sys:System
    EquivalentTo: State or (expr:isOperandOf some StateVariable)

  Class: ActiveState
    EquivalentTo: State and expr:True

  Class: SuperState
    EquivalentTo: State and StateMachine
```

Listing 3.20: The finite state machines ontology (fsm).

Statuses

The concept of "orthogonal regions" of UML state diagrams is similar to *statuses* in our ontology: see listing 3.21. We say that a `Status` is a container of `States` of which there can only be zero or one `ActiveState` at any time. It follows that, since there can never be two or more active states at the time of inferencing, the states of a status must logically exclude each other. For instance, the states

$$\begin{aligned} movingForward &= velocity > window \\ movingBackwards &= velocity < (-window) \\ standstill &= \neg(movingForward \vee movingBackwards) \end{aligned}$$

of some *VelocityStatus* can never be active at the same time. Logical exclusion ($\rightarrow \neg$) is represented by the `excludes` relationship of the *expressions* ontology. Verification of the logical exclusion of states, was one of the main reasons why we attempted to implement inference rules for the `excludes` relationship. Because of these inference rules, our ontology is able to infer the logical exclusion of simple expressions such as the above example (in this case, the inference rules fired are $(x > y) \rightarrow \neg(x < -y)$, $(\neg(a \vee b)) \rightarrow \neg a)$, and $(\neg(a \vee b)) \rightarrow \neg b)$). However, as will be elaborated in the evaluation chapter, executing these inference rules for a huge amount of operations is very costly, and it is still able to spot only the most simple logical exclusions. The added value of adding rules to infer logical exclusion is therefore questionable.

```
  Class: Status
    EquivalentTo: cont:contains exactly 1 ActiveState
    SpinRule:
      CONSTRUCT {
```

```
      _:b0 a spin:ConstraintViolation .
      _:b0 spin:violationLevel spin:Warning .
      _:b0 rdfs:label ?msg
    } WHERE {
      ?this cont:contains ?state1 .
      ?this cont:contains ?state2 .
      FILTER( ?state1 != ?state2 ) .
      FILTER NOT EXISTS { ?state1 expr:excludes ?state2 }
      BIND(fn:concat("Status warning: ", ?state1,
                     " may not exclude ", ?state2) AS ?msg)
    }


ObjectProperty: hasStatus
  SubPropertyOf: sys:hasProperty
  Domain: StateMachine
  Range: Status
```

Listing 3.21: The finite state machines ontology (fsm): statuses.

Transitions

Transitions are defined using the `Implication`, `And`, and `Eventually` operations of the expressions ontology: see listing 3.22. More specifically: a transition from state p to state q for a condition c is defined as a shorthand for the expression $p \wedge c \rightarrow \Diamond q$, saying that p and c eventually lead to q. A *condition* it is not necessarily a *state* of the system, it can be any boolean expression (such as the constant `expr:true`). The shorthand expression for $p \wedge c \rightarrow \Diamond q$ doesn't say much by itself, but it can easily be combined with the $\Box$ operator (*always*) to express the constraints of a system. For instance, suppose that the states *standstill* and *movingForward* are expressed as in the previous example, and additionally a state *stopping* is a boolean that becomes *true* if a "stop" button is pressed. Then we can define a transition:

$$\begin{aligned} transition(transitsFrom &= movingForward, \\ transitsTo &= standstill, \\ hasCondition &= stopping, \\ hasInterval &= [0, 3s]) \end{aligned}$$

$$\leftrightarrow \ ((movingForward \wedge stopping) \rightarrow \Diamond_{[0,3s]} standstill)$$

As will be seen in 3.3.13, we can convert this transition into a *constraint* of the system by stating that $\Box(transition)$, or in other words: "always if moving forward and the stop button is pressed, then the system must be standing still within three seconds".

```
Class: Transition
  SubClassOf: expr:Operation
  SubClassOf: transitsFrom some State
                and transitsTo some State
                and hasCondition some expr:BoolExpression
                and expr:hasInterval some expr:Interval
```

```
7     EquivalentTo: expr:isEquivalentTo some
8                     expr:Implication
9                       and expr:hasLeftOperand some
10                        (expr:And
11                           and expr:hasLeftOperand some State
12                           and expr:hasRightOperand some expr:
                                  BoolExpression)
13                      and expr:hasRightOperand some
14                        (expr:Eventually
15                           and expr:hasOperand some State)
16    SpinRule:
17      CONSTRUCT {
18        ?this :transitsFrom ?fromState .
19        ?this :transitsTo   ?toState .
20        ?this :hasCondition ?condition .
21        ?this expr:hasInterval ?interval
22      } WHERE {
23        ?this expr:isEquivalentTo ?implication .
24        ?implication expr:hasOperator expr:implies .
25        ?implication expr:hasLeftOperand ?and .
26        ?implication expr:hasRightOperand ?eventually .
27        ?and expr:hasOperator expr:and .
28        ?and expr:hasLeftOperand ?fromState .
29        ?and expr:hasRightOperand ?condition .
30        ?eventually expr:hasOperator expr:eventually .
31        ?eventually expr:hasOperand ?toState .
32        ?eventually expr:hasInterval ?interval
33      }
34
35  ObjectProperty: hasTransition
36    SubPropertyOf: sys:hasProperty
37    Domain: StateMachine
38    Range: Transition
39
40  ObjectProperty: isTransitionOf
41    InverseOf: hasTransition
42
43  ObjectProperty: transitsFrom
44    Domain: Transition
45    Range: State
46
47  ObjectProperty: transitsTo
48    Domain: Transition
49    Range: State
50
51  ObjectProperty: hasCondition
52    Domain: Transition
53    Range: expr:BoolExpression
```

Listing 3.22: The finite state machines ontology (fsm): transitions.

3.3.10 Colors

The *colors* ontology is a very simple ontology that defines colors as hexadecimal values. We did not formally define the exact semantics of the bit sequence (i.e. 8 bits for the red, green, and blue channel, respectively), but such extensions to the ontology can still be added in the future. We needed to create this ontology, to be able to specify the colors of the wires of the electrical systems. Table 3.5 lists the "predefined" colors and their hexadecimal value, and listing 3.23 shows a small excerpt of the ontology.

Table 3.5: Colors defined by the *colors* ontology.

Individual	*Hex value*	*Individual*	*Hex value*	*Individual*	*Hex value*
aqua	00FFFF	lime	00FF00	red	FF0000
black	000000	maroon	800000	silver	C0C0C0
blue	0000FF	navy	000080	teal	008080
brown	A52A2A	olive	808000	turquoise	40E0D0
fuchsia	FF00FF	orange	FFA500	violet	EE82EE
gray	808080	pink	FFC0CB	white	FFFFFF
green	008000	purple	800080	yellow	FFFF00

```
Ontology <http://www.mercator.iac.es/onto/metamodels/colors>

Class: Color

DataProperty: hasHexValue

Individual: aqua
  Facts: hasHexValue "00FFFF"

Individual: black
  Facts: hasHexValue "000000"

Individual: blue
  Facts: hasHexValue "0000FF"

# ... 18 more
```

Listing 3.23: The colors ontology (colors).

3.3.11 Geometry

An attempt was made to model several geometric concepts and their relations, resulting in an extensive *geometry* ontology of almost 1 500 lines of code. The purpose was to add the ability to our framework to express the geometric relations between components of the control system, in particular by modeling a coordinate system for each component and the possible transformations (rotation, translation, scaling, ...) of those coordinate systems. Ultimately, we wanted a reasoner to be able to infer, for instance, if the rotation axis of a mirror and a motor shaft are parallel or not (regardless of the number of gears in between), if they rotate at the same or a different velocity and/or direction, if the motor shaft rotates "clockwise" or "anti-clockwise" compared to the motor frame for positive velocities, and so on.

In our ontology, we represented concepts such as shapes (`Shape`, `LinearShape`, `PlanarShape`, `Point`, `SpatialShape`, `Axis`, `Line`, `LineSegment`, `Vector`, `PointVector`, `UnitVector`, `Direction`, `CoordinateSystem`, `Origin`), transformations (`Transformation`, `FixedTransformation`, `ComposedTransformation`, `Rotation`), and their relationships such as `intersects`, `isParallelTo`,

`isNotParallelTo`, `isOrthogonalTo`, `isPointOf`, `isFixedTo`, etc. We were able to model several rules of geometry using SPIN rules: see for instance the definition of a `LinearShape` in listing 3.24:

```
Ontology <http://www.mercator.iac.es/onto/metamodels/geometry>
  Import: <http://www.mercator.iac.es/onto/metamodels/mathematics>

Class LinearShape
  SubClassOf: Shape
  SubClassOf: hasDirection exactly 1 Direction
  SpinRule:
    CONSTRUCT {
      ?this geom:isOrthogonalTo ?shape2 .
    } WHERE {
      ?this geom:isParallelTo ?shape1 .
      ?shape1 geom:isOrthogonalTo ?shape2
    }
  SpinRule:
    CONSTRUCT {
      ?this geom:isNotParallelTo ?shape2 .
    } WHERE {
      ?this geom:isParallelTo ?shape1 .
      ?shape1 geom:isNotParallelTo ?shape2
    }
  # (and so on...)
```

Listing 3.24: The geometry ontology (geom).

As will be elaborated in chapter 5 – *Evaluation*, trivial relationships such as parallelism, non-parallelism, orthogonality, collinearity, etc. are often fairly straightforward to model. But as soon as numerical computations are needed (e.g. to determine orthogonality of two vectors by performing a dot product), it turns out that – unsurprisingly – the SPIN rules become very tedious to write, slow to execute, and dependent on extensions by other languages such as javascript. In the end, we decided to not add more SPIN rules to the *geometry* ontology since their usefulness, for our application, turned out to be very limited. While several ontologies depend on the *geometry* ontology, only a small percentage of the defined classes and relationships are actually reused. The full ontology can be accessed online via its URI: `http://www.mercator.iac.es/onto/metamodels/geometry`.

3.3.12 Control systems

To describe control systems in a very abstract way, we considered it necessary to create a *control systems* ontology: see listing 3.25. This ontology is based on the concepts of a *controller*, *actuator*, *sensor*, *controlled system*, and the information (called *signals*) they exchange. For instance, a controller is something that controls some system property, by producing a controller signal. An actuator consumes this controller signal and produces in turn an actuator signal, which is consumed by the controlled system. A sensor finally senses a property of a system, and produces a sensor signal which may be consumed by a controller in case of a closed-loop control system.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/controlsystems>
  Import: <http://www.mercator.iac.es/onto/metamodels/mathematics>

Class: Controller
  EquivalentTo: controls some sys:Property
  SubClassOf: produces some ControllerSignal

Class: Actuator
  EquivalentTo: (consumes some ControllerSignal) and (produces some
      ActuatorSignal)

Class: Sensor
  EquivalentTo: senses some sys:Property
  SubClassOf: produces some SensorSignal

Class: ControlledSystem
  EquivalentTo: sys:hasProperty some (isControlledBy min 1)
  SubClassOf: consumes some ActuatorSignal

ObjectProperty: controls
  Characteristics: Asymmetric, Irreflexive
  Domain: Controller
  Range: sys:Property

ObjectProperty: isControlledBy
  InverseOf: controls

ObjectProperty: senses
  Characteristics: Asymmetric, Irreflexive
  Domain: Sensor
  Range: sys:Property

ObjectProperty: isSensedBy
  InverseOf: senses
```

Listing 3.25: The control systems ontology (ctrl).

Signals

The relevant signals are shown in listing 3.26. Subsumption is not asserted explicitly, but a reasoner can infer the *is-a* relation (e.g. `ControllerSignal rdfs:subClassOf Signal`) based on the definitions below and the knowledge that a `Controller` (or `Actuator` or `Sensor`) *is-a* `owl:Thing`.

```
Class: Signal
  EquivalentTo: qudt:Quantity and :isProducedBy some owl:Thing

Class: ControllerSignal
  EquivalentTo: qudt:Quantity and :isProducedBy some Controller

Class: ActuatorSignal
  EquivalentTo: qudt:Quantity and :isProducedBy some Actuator

Class: SensorSignal
  EquivalentTo: qudt:Quantity and :isProducedBy some Sensor
```

Listing 3.26: The control systems ontology (ctrl): signals.

Produces and consumes

`Produces` and `consumes` are irreflexive, asymmetric and disjoint relationships, to relate a system with a signal (defined as a produced quantity): see listing 3.27. The disjointness of `produces` and `consumes` comes from the fact that if A produces B, then A consumes *minus* B[8]. For instance, if an electric motor (which is an actuator according to our *electricity* ontology, see 3.3.16) is braking and thus consuming mechanical power P (by converting it back to electrical power and heat per time unit), then it is producing $-P$. As seen in listing 3.27, we can express this logic using SPIN rules.

```
ObjectProperty: produces
  Characteristics: Irreflexive, Asymmetric
  DisjointWith: consumes
  Domain: sys:System
  Range: Signal

ObjectProperty: consumes
  Characteristics: Irreflexive, Asymmetric
  Domain: sys:System
  Range: Signal

Class: Producer
  EquivalentTo: produces some Signal
  SpinRule:
    CONSTRUCT {
      ?this :consumes ?negativeSignal
    } WHERE {
      ?this :produces ?signal .
      ?negativeSignal expr:hasOperand ?signal .
      ?negativeSignal expr:hasOperator math:unaryMinus
    }

Class: Consumer
  EquivalentTo: consumes some Signal
  SpinRule:
    CONSTRUCT {
      ?this :produces ?negativeSignal
    } WHERE {
      ?this :consumes ?signal .
      ?negativeSignal expr:hasOperand ?signal .
      ?negativeSignal expr:hasOperator math:unaryMinus
    }
```

Listing 3.27: The control systems ontology (ctrl): producers and consumers.

3.3.13 Development

According to our *development* ontology, system development starts by defining a *project*: see listing 3.28. We cannot give a complete definition of a development `Project`, but we constrain its meaning by saying that it must have a `Purpose` and at least a `Concept` or `Design` or `Implementation`. A `Concept` is formally

[8]Note: B and minus B always represent different ("disjoint") quantities, even if numerically they can both be equal to zero

defined as a model of something: one that is proven to have no *parts*; it can only have *properties* or other *elements*. The idea is that a concept represents a system as a whole: a black box that is not yet broken down into parts. Therefore, any constraints defined at the conceptual level can only consider the system as a whole, via "top level" system properties. A `Design` on the other hand is simply the realization of this concept; and an `Implementation` is the realization of a design. Designs and implementations do not have the same restriction as concepts: they can have both *properties* and *parts*.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/development>
  Import: <http://www.mercator.iac.es/onto/metamodels/containers>
  Import: <http://www.mercator.iac.es/onto/metamodels/finitestatemachines>
  Import: <http://www.mercator.iac.es/onto/metamodels/models>
  Import: <http://www.mercator.iac.es/onto/metamodels/organizations>

Class: Project
  SubClassOf: mod:hasPurpose some mod:Purpose
  SubClassOf: sys:hasPart some (Concept or Design or Implementation)

Class: Concept
  EquivalentTo: mod:Model and (sys:hasPart max 0)

Class: Design
  EquivalentTo: sys:realizes some Concept

Class: Implementation
  EquivalentTo: sys:realizes some Design
```

Listing 3.28: The development ontology (dev).

Requirements and constraints

A concept (i.e. a black box model of a system) may list a number of `Requirements`, which represent constraints that must be satsified, of the system that it tries to model: see listing 3.29. As illustrated in appendix A (e.g. see listing A.2), we mostly express these requirements using natural language, because our ontologies provide much too little expressiveness to express realistic conceptual requirements such as *"The telescope shall be able to move between any two targets on the sky within 3 minutes"*. However, we can approximate these requirements by formal `Constraints`: boolean expressions about the system that *always* have to be true. Hence, we can reuse the temporal logic operation `Always` ($\Box$) to express a constraint. These formal constraints `represent` the natural language requirements, which at their turn `represent` the "real" constraints of the system. If a constraint evaluates to *true*, then we say that it `satisfies` the requirement which it represents. If a requirement – or any of its derived requirements, see next paragraph – is not satisfied in the closed world, we raise a constraint violation.

```
Class: Requirement
  EquivalentTo: mod:represents some Constraint
```

```
    SpinConstraint:
      CONSTRUCT {
        _:b0 a spin:ConstraintViolation .
        _:b0 rdfs:label "Requirement is not satsified!"
      } WHERE {
        {
          FILTER NOT EXISTS { ?this :isSatisfiedBy ?x }
        }
        UNION
        {
          FILTER NOT EXISTS { ?req :isDerivedFrom ?this .
                              ?req :isSatisfiedBy ?y    }
        }
      }

  Class: Constraint
    SubClassOf: expr:Constant
    SubClassOf: expr:Always and
                      (expr:hasOperand some
                         (sys:isPropertyOf some sys:System))
    SubClassOf: mod:represents some Requirement
    SpinRule:
      CONSTRUCT {
        ?this :satisfies ?req
      } WHERE {
        ?this sys:represents ?req .
        ?this expr:hasValue true
      }

  ObjectProperty: satisfies
    Characteristics: Irreflexive
    Domain: sys:System
    Range: Requirement

  ObjectProperty: isSatisfiedBy
    InverseOf: satisfies
```

Listing 3.29: The development ontology (dev): requirements and constraints.

Derivation and refinement

Our ontology defines two special relationships between requirements: derivation and refinement. Derivation means that one requirement is split up in multiple more "narrow" requirements via the `derives` relationship: see listing 3.30. As seen in the previous paragraph, it means that a requirement can only be satisfied if *all* derived requirements are satisfied. For instance, the requirement

"The telescope axes shall move at a maximum velocity of 3 °/s"

can only be satisfied if the derived requirements

"The azimuth axis shall move at a maximum velocity of 3 °/s"
"The elevation axis shall move at a maximum velocity of 3 °/s"

are both satisfied. Refinement, on the other hand, means that a single requirement is able to satisfy the requirement that it refines. For instance, the requirement

"The elevation axis shall move at a maximum velocity of 2.5 °/s"

refines the last mentioned requirement. Refinement holds an implication: if the refining requirement is satisfied, then it follows that the refined requirement is satisfied as well.

```
Class: RefinedRequirement
  EquivalentTo: isRefinedBy min 1
  SpinRule:
    CONSTRUCT {
      ?req :satisfies ?this .
    } WHERE {
      ?this :isRefinedBy ?req .
      ?x :satisfies ?req .
    }

ObjectProperty: derives
  Characteristics: Irreflexive
  Domain: Requirement
  Range: Requirement

ObjectProperty: isDerivedFrom
  InverseOf: derives

ObjectProperty: refines
  Characteristics: Irreflexive
  Domain: Requirement
  Range: Requirement

ObjectProperty: isRefinedBy
  InverseOf: refines

ObjectProperty: verifies
  Domain: Test
  Range: Requirement
  SubPropertyChain: tests o mod:represents
```

Listing 3.30: The development ontology (dev): derivation and refinement.

Assertions

A requirement is satisfied if the corresponding constraint evaluates to *true*. Sometimes, this evaluation may be performed by reasoning: for example, if a constraint is expressed as a `LessThan` operation between the known and the maximum cost of a system, then the reasoner is able to infer the truth of this expression due to the SPIN rules that we added to the `LessThan` operation. Very often however, insufficient knowledge is available to the reasoner, and the truth of the constraint must be asserted in another way. For this reason, we defined an `Assertion`: see listing 3.31. An assertion may be, for instance, a document (e.g. a data sheet by a vendor) or a test result. As can be seen in the listing, a few very simple testing primitives were added to the *development* ontology. The idea is to allow the quick expression of tests while the system is being modeled, by saying that some `Test` individual `tests` a constraint. If at a later stage the test is performed and the result is a simple `pass` or `fail`, then this information can be added to the model. In case of a passed test, the

reasoner will infer that the constraint is asserted, and that the corresponding requirement is satisfied.

```
Class: Assertion
  EquivalentTo: asserts some Constraint

ObjectProperty: asserts
  Range: expr:True

Class: Test
  EquivalentTo: tests min 1
  SpinRule:
    CONSTRUCT {
      ?this :asserts ?x
    } WHERE {
      ?this :tests ?x .
      ?this :hasResult :pass
    }

Class: TestResult

Individual: pass
  Types: TestResult

Individual: fail
  Types: TestResult
  DifferentFrom: pass

ObjectProperty: tests
  Domain: Test

ObjectProperty: hasResult
  Domain: Test
  Range: TestResult
  Characteristics: Asymmetric, Irreflexive
```

Listing 3.31: The development ontology (dev): assertions.

Of course, we can also express directly that a natural language requirement is satisfied by some model element. In this case, we do not even try to formulate a formal constraint for the requirement: we simply say that some model element satisfies the requirement. This may be useful to satisfy specific requirements such as *"The cabinet shall have an emergency stop button"*, in which case an emergency button individual may be linked directly to the requirement via the `satisfies` relationship.

3.3.14 Manufacturing

To be able to model manufacturers and their products, we created a very simple *manufacturing* ontology. Not many formal constraints could be put on its definitions, but the vocabulary is sufficient to express that a product has an ID that is unique to the manufacturer, and that is manufactured by a manufacturer – an organization that manufactures at least one product. A product may realize multiple systems, but it may only have one manufacturing type. This constraint can be helpful to verify if, for instance, an individual of a mechanical model and an individual of an electrical model both represent the same device. If the

individuals of both models are said to be the same (by an `owl:sameAs` relation) but their type does not correspond, then a reasoner capable of interpreting the `hasType max 1` cardinality restriction will raise an error.

```
Class: Manufacturer
  EquivalentTo: org:Organization and manufactures min 1
  SpinRule:
    CONSTRUCT {
      ?product1 owl:sameAs ?product2
    WHERE {
      ?this :manufactures ?product1 .
      ?this :manufactures ?product2 .
      FILTER(?product1 != ?product2) .
      ?product1 :hasId ?id .
      ?product2 :hasId ?id
    }

Class: Product
  SubClassOf: dev:Design or dev:Implementation
  SubClassOf: isManufacturedBy some Manufacturer
  SubClassOf: hasType max 1
  SpinConstraint:
    CONSTRUCT {
      _:b0 a spin:ConstraintViolation .
      _:b0 spin:violationRoot ?this .
      _:b0 spin:violationLevel spin:Error .
      _:b0 rdfs:label "A product must have a unique ID!" }
    WHERE {
      ?this :hasId ?id .
      ?this :hasId ?otherId .
      FILTER( ?id != ?otherId )
    }

ObjectProperty: manufactures
  Domain: Manufacturer
  Range: Product

ObjectProperty: isManufacturedBy
  InverseOf: manufactures

DataProperty: hasId
  Domain: Product
  Range: xsd:string

ObjectProperty: hasType
  SubPropertyOf: sys:realizes
  Domain: Product
  Range: Product

ObjectProperty: isTypeOf
  InverseOf: hasType
```

Listing 3.32: The manufacturing ontology (man).

3.3.15 Mechanics

As explained in the beginning of this chapter, our framework has been implemented with a particular goal in mind: the development of a new control system for the Mercator telescope. While all the software aspects and most of the electrical aspects of the original system needed to be revised, most of the

mechanical aspects (such as the transmission ratios and angular velocities of the rotating components of the system) remained the same. This is the main reason why our *mechanics* ontology is not very expressive, as we did not anticipate to model many of the mechanical aspects of the Mercator Telescope. The purpose of the *mechanics* ontology is therefore limited to the representation of parts and assemblies, and of systems that involve rotation (such as very simple gear transmissions and motors).

Assemblies and parts

Assemblies and their parts can easily be represented by extending the `sys:hasPart` relationship to the mechanical domain: see listing 3.33. The mechanical `hasPart` relationship does not "inherit" the transitivity of the systems `hasPart` relationship, allowing us to "browse" a mechanical model hierarchically.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/mechanics>
  Import: <http://www.mercator.iac.es/onto/metamodels/controlsystems>
  Import: <http://www.mercator.iac.es/onto/metamodels/finitestatemachines>
  Import: <http://www.mercator.iac.es/onto/metamodels/geometry>

ObjectProperty: hasPart
  SubPropertyOf: sys:hasPart
  Characteristics: Asymmetric, Irreflexive
  Domain: Assembly
  Range: Part

ObjectProperty: isPartOf
  InverseOf: hasPart

Class: Assembly
  EquivalentTo: hasPart some Part

Class: Part
  EquivalentTo: isPartOf some Assembly
```

Listing 3.33: The mechanics ontology (mech): assemblies and parts.

Motors and loads

We defined a mechanical `Motor` as something that produces some mechanical `Power`, and a mechanical `Load` as something that consumes some mechanical `Power`: see listing 3.34. Producing and consuming were defined earlier by the *controllers* ontology. We expressed mechanical `Power` using the QUDT ontology (described in 3.3.4), which defines 31 "mechanics" quantity kinds such as `Mass`, `Power`, `Torque`, etc.

```
Class: Motor
  EquivalentTo: ctr:produces some Power

```

```
Class: Load
  EquivalentTo: ctrl:consumes some Power

Class: Power
  SubClassOf: qudt:Quantity
  EquivalentTo: qudt:quantityKind value qudt-quantity:Power
```

Listing 3.34: The mechanics ontology (mech): motors and loads.

Rotary motors and loads

Kinematic properties of motors and loads are based on the *geometry* ontology: the idea is that a motion of an object is defined as a geometric transformation. For instance, the "velocity of a rotary motor" is defined as the velocity of the `geom:Rotation` of a reference frame of the motor shaft, around an axis of a reference frame of the motor housing. Likewise, the "position" of an encoder is a property of the rotation of the reference frame of the encoder shaft, around an axis of the reference frame of the encoder housing. The motion of a part is thus always relative to another part: there's no such thing as "the" velocity (or acceleration, torque, position, ...) of a certain part of a gear train. Reference frames are instances of the `geom:CoordinateSystem` class. As shown in 3.35, we defined the concepts `Stator` and `Rotor` to describe the "direction" of the rotation more conveniently: a rotor has a coordinate system that rotates around an axis of the stator coordinate system. Of course, this direction of rotation is purely conventional: the housing of a rotary motor can be a stator or a rotor, depending on how the rotation is expressed.

```
Class: RotaryMotor
  EquivalentTo: Motor and hasStator exactly 1 and hasRotor exactly 1

Class: RotaryLoad
  EquivalentTo: Load and hasStator exactly 1 and hasRotor exactly 1

Class: Stator
  EquivalentTo: mech:Part
                and (geom:hasCoordinateSystem some
                      (geom:hasAxis some
                        (geom:isRotationAxisOf some geom:Rotation)))

Class: Rotor
  EquivalentTo: mech:Part
                and (geom:hasCoordinateSystem some
                      (expr:isOperandOf some geom:Rotation))

ObjectProperty: hasStator
  SubPropertyOf: hasPart
  Range: Stator

ObjectProperty: hasRotor
  SubPropertyOf: hasPart
  Range: Rotor
```

Listing 3.35: The mechanics ontology (mech): rotary motors and loads.

Transmissions and rotary transmissions

A mechanical transmission is defined as something that both produces and consumes some mechanical power: see listing 3.36. Rotary transmissions have one stator, two rotors, and three possible rotations: one between each rotor and stator, and one between the two rotors. The latter rotation is used to differentiate the two rotors: it is defined as the rotation of the *output rotor* around an axis of a coordinate system of the *input rotor* (which thus acts as a stator for this rotation, as can be inferred by a reasoner).

```
53  Class: Transmission:
54    EquivalentTo: (ctrl:consumes some Power)
55                  and (ctrl:produces some Power)
56    SubClassof: hasTransmissionRatio exactly 1
57    SubClassof: hasTransmissionEfficiency exactly 1
58
59  Class: RotaryTransmission
60    EquivalentTo: Transmission
61                  and :hasRotor exactly 2
62                  and :hasStator exactly 1
63    SpinRule:
64      CONSTRUCT {
65        ?this :hasInputRotor ?inputRotor .
66        ?this :hasOutputRotor ?outputRotor
67      } WHERE {
68        ?this :hasRotor ?inputRotor .
69        ?this :hasRotor ?outputRotor .
70        FILTER(?inputRotor != ?outputRotor) .
71        ?inputRotor  geom:hasCoordinateSystem ?inputRotorCS .
72        ?outputRotor geom:hasCoordinateSystem ?outputRotorCS .
73        ?rotation expr:hasOperand ?outputRotorCS .
74        ?inputRotorCS geom:hasAxis/geom:isRotationAxisOf ?rotation
75      }
76
77  ObjectProperty: hasInputRotor
78    SubPropertyOf: hasRotor
79    Domain: Assembly
80    Range: Rotor
81
82  ObjectProperty: hasOutputRotor
83    SubPropertyOf: hasRotor
84    Domain: Assembly
85    Range: Rotor
```

Listing 3.36: The mechanics ontology (mech): tranmissions and rotary transmissions.

As already visible in the definition of `Transmission`, we defined two characteristics of transmissions: the `TransmissionRatio` and `TransmissionEfficiency`, see listing 3.37. The former is defined as the ratio between the input velocity and output velocity, the latter as the ratio between the input power and output power. The SPIN rules unambiguously fix the meaning of the ratios. If quantity values and corresponding numerical values are available, then the reasoner is able to infer "missing" velocities and powers of the transmissions based on the transmission ratios and efficiencies – or vice versa.

```
Class: TransmissionRatio
  SubClassOf: qudt:Quantity
  SubClassOf: qudt:quantityKind value qudt-quantity:DimensionlessRatio
  SubClassOf: math:Division
  SpinRule:
    # rule only matching rotary tranmsissions:
    CONSTRUCT {
      ?this expr:hasLeftOperand ?inputVelocity .
      ?this expr:hasRightOperand ?outputVelocity
    } WHERE {
      ?tranmsission :hasTransmissionRatio ?this .
      ?transmission :hasInputRotor/geom:hasCoordinateSystem ?inputRotorCS
          .
      ?transmission :hasOutputRotor/geom:hasCoordinateSystem ?
          outputRotorCS .
      ?transmission :hasStator ?stator .
      ?inputRotation expr:hasOperand ?inputRotorCS .
      ?outputRotation expr:hasOperand ?outputRotorCS .
      ?inputRotation geom:hasRotationAxis ?ax .
      ?outputRotation geom:hasRotationAxis ?ax .
      ?stator geom:hasCoordinateSystem/geom:hasAxis ?ax .
      ?inputRotation geom:hasAngularVelocity ?inputVelocity .
      ?outputRotation geom:hasAngularVelocity ?outputVelocity
    }

Class: TransmissionEfficiency
  SubClassOf: qudt:Quantity
  SubClassOf: qudt:quantityKind value qudt-quantity:DimensionlessRatio
  SubClassOf: math:Division
  SpinRule:
    CONSTRUCT {
      ?this expr:hasLeftOperand ?inputPower .
      ?this expr:hasRightOperand ?outputPower
    } WHERE {
      ?transmission :hasTransmissionEfficiency ?this .
      ?transmission ctrl:produces ?outputPower .
      ?transmission ctrl:consumes ?inputPower .
      ?outputPower rdf:type :Power .
      ?inputPower rdf:type :Power
    }
```

Listing 3.37: The mechanics ontology (mech): transmission ratio and efficiency.

Fixed parts

Finally, we defined an `isFixedTo` relationship, which is a subproperty of the `geom:isFixedTo` relationship of the *geometry* ontology: see listing 3.38. The latter relationship says that there is a `geom:FixedTransformation` between two shapes that are fixed to each other. Fixed transformations have constant zero angular velocity and constant zero linear velocity, by definition.

```
ObjectProperty: isFixedTo
  Domain: Part
  Range: Part
  Characteristics: Transitive, Symmetric
  SubPropertyOf: geom:isFixedTo
```

Listing 3.38: The mechanics ontology (mech): fixed parts.

3.3.16 Electricity

To be able to express electrical properties of our embedded systems, we created an *electricity* ontology: see listing 3.39. It reuses the `produces` and `consumes` relationships of the *control systems* ontology to define electric `Consumer`s and `Producer`s as things that consume (or produce, respectively) some electric `Power`. All producers and consumers are called `Device`s. Only if something is both a producer and a consumer, then we call it a `Converter`. An electric `Configuration` finally is not fully defined; we only state that it must contain at least one electric device.

```
1  Ontology <http://www.mercator.iac.es/onto/metamodels/electricity>
2    Import: <http://www.mercator.iac.es/onto/metamodels/mechanics>
3    Import: <http://www.mercator.iac.es/onto/metamodels/iec61131>
4    Import: <http://www.mercator.iac.es/onto/metamodels/colors>
5
6  Class: Power
7    SubClassOf: Quantity
8    EquivalentTo: qudt:quantityKind value qudt-quantity:ElectricPower
9
10 Class: Consumer
11   EquivalentTo: ctrl:consumes some Power
12
13 Class: Producer
14   EquivalentTo: ctrl:produces some Power
15
16 Class: Device
17   EquivalentTo: Consumer or Producer
18
19 Class: Converter
20   EquivalentTo: Consumer and Producer
21
22 Class: Configuration
23   SubClassOf: cont:contains some Device
```

Listing 3.39: The electricity ontology (elec).

Actuators, sensors, motors, ...

The electric systems we need to describe can be classified using the terminology of the *control systems* ontology: see listing 3.40. We first specialize the *actuator* and *sensor* concepts to the electrical domain, by stating that they are electrical consumers. An electric `Motor` is then an electric actuator that produces some mechanical power, and an `Encoder` is an electric sensor that senses some position. An electric `Drive` and `PowerSupply` are both an electric converter and controller: the former controls mechanical power, and the latter controls the electric power that it produces.

```
24 Class: Actuator
25   EquivalentTo: ctrl:Actuator and Consumer
26
27 Class: Sensor
28   EquivalentTo: ctrl:Sensor and Consumer
```

```
29
30  Class: Motor
31    EquivalentTo: Actuator and ctrl:produces some mech:Power
32
33  Class: Encoder
34    EquivalentTo: Sensor and ctrl:senses some geom:Position
35
36  Class: Drive
37    EquivalentTo: Converter and ctrl:controls some mech:Power
38
39  Class: PowerSupply
40    EquivalentTo: Converter and ctrl:controls some Power
41    SpinRule:
42      CONSTRUCT {
43        ?this ctrl:controls ?power
44      } WHERE {
45        ?this ctrl:produces ?power .
46        ?power rdf:type elec:Power
47      }
```

Listing 3.40: The electricity ontology (elec): actuators, sensors, motors, encoders, drives, power supplies.

I/O modules

I/O modules could not be fully defined via an `EquivalentTo` statement: we only stated that it is a kind of electrical device, that it has at least one channel, and that it has a software interface (i.e. a `sys:Interface` that only consist of software variables, see 3.3.17). If a channel of an I/O module also has a software interface, then this software interface is part of the software interface of the I/O module: see the SPIN rule at listing 3.41.

```
48  Class: IoModule
49    SubClassOf: Device
50    SubClassOf: hasChannel min 1
51    SubClassOf: sys:hasInterface some soft:Interface
52    CONSTRUCT {
53      ?interface soft:hasVariable ?variable
54    } WHERE {
55      ?this sys:hasInterface ?interface .
56      ?this :hasChannel/sys:hasInterface ?channelInterface .
57      ?channelInterface soft:hasVariable ?variable
58    }
59
60  Class: Channel
61    SubClassOf: hasTerminal min 1
62
63  ObjectProperty: hasChannel
64    SubPropertyOf: sys:hasPart
65    Domain: Device
66    Range: Channel
67
68  ObjectProperty: hasTerminal
69    SubPropertyOf: hasPart
70    Domain: Device
71    Range: Terminal
```

Listing 3.41: The electricity ontology (elec): I/O modules.

Conductors, insulators, and electric connections

The meaning of conductors and insulators is expressed using the `isConnectedTo` relationship, which represents an electrical connection. Electrical connections can only take place between `Conductor`s: mechanical parts which may only consist of other electrically connected conductors. `Insulator`s on the other hand cannot be electrically connected to something else, and they can only consist of other insulators. A `Connection` finally `connects` two conductors: it implies `isConnectedTo` relations between all connected conductors (see listing 3.42).

```
72  Class: Conductor
73    SubClassOf: mech:Part
74    SpinRule:
75      CONSTRUCT {
76        ?this :isConnectedTo ?part
77      } WHERE {
78        ?this mech:hasPart ?part .
79      }
80
81  Class: Insulator
82    DisjointWith: Conductor
83    SubClassOf: (isConnectedTo max 0) and (sys:hasPart only Insulator)
84
85  ObjectProperty: isConnectedTo
86    Characteristics: Symmetric
87    Domain: Conductor
88    Range: Conductor
89
90  Class: Connection
91    SubClassOf: connects min 2
92    SpinRule:
93      CONSTRUCT {
94        ?x :isConnectedTo ?y .
95      } WHERE {
96        ?this :connects ?x .
97        ?this :connects ?y
98      }
99
100 ObjectProperty: connects
101   Characteristics: Asymmetric, Irreflexive
102   Domain: Connection
103   Range: Conductor
```

Listing 3.42: The electricity ontology (elec): conductors, insulators, electric connections.

Wires, cables, and cable assemblies

A `Wire` is modeled as an electric connection, consisting of a single conductor (which may or may not be covered by an insulating layer). A wire may be part of a `Cable`, and a cable may be part of a `CableAssembly`. A cable assembly represents what is often informally called “a cable”: an assembly of one or two connectors and the actual cable. See listing 3.43.

```
Class: Wire
  SubClassOf: Connection and mech:hasPart exactly 1 Conductor

Class: Cable
  SubClassOf: mech:hasPart some Wire

Class: CableAssembly
  SubClassOf: mech:hasPart some Cable
```

Listing 3.43: The electricity ontology (elec): wires, cables, cable assemblies.

Terminals and connectors

An electrical `Terminal` is defined as a conductor that is mechanically fixed to some mechanical part. If this part turns out to be a conductor, then it can be inferred that the terminal and the part are electrically connected to each other. We also constrained a `Connector` by saying that it must have an ordered list of terminals. Particular for a connector is that if it is *joined* with another connector, then all terminals with corresponding numbers are mechanically fixed (and, hence, electrically connected) to each other. We implemented this rule using SPIN, see listing 3.44.

```
Class: Terminal
  SubClassOf: Conductor and mech:isFixedTo some mech:Part
  SpinRule:
    CONSTRUCT {
      ?this :isConnectedTo ?x
    } WHERE {
      ?this mech:isFixedTo ?x .
      ?x rdf:type/rdfs:subClassOf* :Conductor
    }

Class: Connector
  SubClassOf: sys:hasPart some (List and cont:contains some Terminal)
  SpinRule:
    CONSTRUCT {
      ?thisTerminal mech:isFixedTo ?otherTerminal
    } WHERE {
      ?this :isJoinedWith ?other .
      ?this sys:hasPart/cont:contains ?thisTerminal .
      ?thisTerminal rdf:type/rdfs:subClassOf* :Terminal .
      ?thisTerminal cont:isItemOf/cont:hasNumber ?n .
      ?other sys:hasPart/cont:contains ?otherTerminal .
      ?otherTerminal rdf:type/rdfs:subClassOf* :Terminal .
      ?otherTerminal cont:isItemOf/cont:hasNumber ?n
    }

ObjectProperty: isJoinedWith
  SubPropertyOf: isFixedTo
  Characteristics: Symmetric
  Domain: Connector
  Range: Connector
```

Listing 3.44: The electricity ontology (elec): terminals and connectors.

Connector gender

A very simple verification can be implemented for an electrical model, if the genders of the connectors are specified. Listing 3.45 shows the three possible values of the `Gender` class: `male`, `female`, and `hybrid`. Even though our ontology does not define the formal meaning of these gender values, it does force us to use the gender values consistently. Because `hasGender` is a functional property, the connector classes (`MaleConnector`, `FemaleConnector`, and `HybridConnector`) are disjoint with each other. For instance, if `M` is a `MaleConnector` and `H` is a `HybridConnector`, then the statement `M isConnectedTo H` leads to an inconsistency since the reasoner will infer that `H` is also a `FemaleConnector`.

```
142  Class: MaleConnector
143    EquivalentTo: Connector and hasGender value male
144    SubClassOf: isJoinedWith only FemaleConnector
145
146  Class: FemaleConnector
147    EquivalentTo: Connector and hasGender value female
148    SubClassOf: isJoinedWith only MaleConnector
149
150  Class: HybridConnector
151    EquivalentTo: Connector and hasGender value hybrid
152    SubClassOf: isJoinedWith only HybridConnector
153
154  Class: Gender
155    EquivalentTo: { male, female, hybrid }
156
157  ObjectProperty: hasGender
158    Range: Gender
159    Characteristics: Functional
160
161  Individual: male
162    Types: Gender
163    DifferentFrom: female, hybrid
164
165  Individual: female
166    Types: Gender
167    DifferentFrom: male, hybrid
168
169  Individual: hybrid
170    Types: Gender
171    DifferentFrom: male, female
```

Listing 3.45: The electricity ontology (elec): connector gender.

Switches

The *electricity* ontology finally defines `Switch`es as "things that have at least one conductive state". A `ConductiveState` was modeled as the state of a state machine: if the state is active then the state machine is a `Conductor`. Similarly, if an `InsulatingState` is active, then the state machine is an `Insulator`. See listing 3.46 for the corresponding SPIN rules. Switches always have at least one conductive state, but not necessarily an insulating state.

```
Class: Switch
  EquivalentTo: sys:hasPart some (fsm:hasState some ConductingState)

Class: ConductingState
  SubClassOf: fsm:State
  SpinRule:
    CONSTRUCT {
      ?stateMachine rdf:type :Conductor
    } WHERE {
      ?this expr:hasValue true .
      ?stateMachine fsm:hasState ?this
    }

Class: InsulatingState
  SubClassOf: fsm:State
  SpinRule:
    CONSTRUCT {
      ?stateMachine rdf:type :Insulator
    } WHERE {
      ?this expr:hasValue true .
      ?stateMachine fsm:hasState ?this
    }
```

Listing 3.46: The electricity ontology (elec): switches.

3.3.17 Software

We created a *software* ontology to provide a fairly abstract (language-independent) vocabulary for designing software, much like UML. In contrast to UML however, our vocabulary is formally defined, it is much less over-constrained, and it is formally extended by ontologies about particular programming languages (such as our *iec61131* ontology) before being converted into actual code. To allow such language extensions, we defined the concepts of `Language` and `LanguageElement`: see listing 3.47.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/containers>
  Import: <http://www.mercator.iac.es/onto/metamodels/documents>
  Import: <http://www.mercator.iac.es/onto/metamodels/mathematics>
  Import: <http://www.mercator.iac.es/onto/metamodels/models>

Class: Language
  SubClassOf: mod:Model
  SubClassOf: sys:hasElement some LanguageElement

Class: LanguageElement
  EquivalentTo: sys:isElementOf some Language
```

Listing 3.47: The software ontology (soft).

Variables

Most software languages support the concept of a `Variable`: a symbolic representation of a memory location. We explicitely modeled the address

of a variable, to allow the definition of pointers and "address-of" functions and operators (such as `ADR` of IEC61131-3, and `&` of C++, respectively). As can be seen in listing 3.48, we implemented two SPIN rules for variables. The first rule says that if variable `A sys:realizes B`, then all "sub-variables" of `A` realize the sub-variables of `B` with the same name (e.g. `A.x sys:realizes B.x`). The second rule says that if variable `A sys:realizes B`, and `B` has a known type, then `A` has the same type. By executing both rules, type information about sub-variables of a structured type will be linked to the corresponding sub-variables of the instances of that type.

```
12  Class: MemoryLocation
13    SubClassOf: hasAddress exactly 1
14
15  DataProperty: hasAddress
16    Domain: MemoryLocation
17
18  Class: Variable
19    EquivalentTo: (mod:represents some MemoryLocation)
20                  and (hasSymbol exactly 1)
21    SubClassOf: hasType some Type
22    SpinRule:
23      CONSTRUCT {
24        ?thisSubVar sys:realizes ?otherSubVar .
25      } WHERE {
26        ?this sys:realizes ?other .
27        ?this :hasVariable ?thisSubVar .
28        ?other :hasVariable ?otherSubVar .
29        ?thisSubVar rdfs:label ?subVarName .
30        ?otherSubVar rdfs:label ?subVarName
31      }
32    SpinRule:
33      CONSTRUCT {
34        ?this :hasType ?type
35      } WHERE {
36        ?this sys:realizes/:hasType ?type
37      }
38
39  DataProperty: hasSymbol
40    SubPropertyOf: rdfs:label
41    Range: xsd:string
42
43  ObjectProperty: hasVariable
44    SubPropertyOf: sys:hasPart
45    Range: Variable
46
47  ObjectProperty: isVariableOf
48    InverseOf: hasVariable
```

Listing 3.48: The software ontology (soft): variables.

Types

A variable may have a `Type`, which puts constraints on the variable. For example, as seen in the previous paragraph, if a type has sub-variables, then all instances of that type must have sub-variables with corresponding name and type information. According to our ontology, a `Variable` represents a broad concept: variables can represent instances of primitive data types, instances

of object-oriented classes, but also instances of methods or even instances of runnable programs. All these variables may have a type. When a type "has" (or sometimes called "declares") other variables, then the latter are called `Attribute`s of that type. Types may also `extend` each other, which represents a kind of realization between two types. The type that extends the other type, is called a `SubType`.

```
Class: Type
  SubClassOf: isTypeOf some Variable

ObjectProperty: hasType
  SubPropertyOf: sys:realizes
  Domain: Variable
  Range: Type

ObjectProperty: isTypeOf
  InverseOf: hasType

Class: Attribute
  EquivalentTo: Variable and sys:isVariableOf exactly 1 Type

ObjectProperty: hasAttribute
  SubPropertyOf: hasVariable
  Range: Attribute

Class: SubType
  SubClassOf: Type
  EquivalentTo: extends some Type

ObjectProperty: extends
  SubPropertyOf: sys:realizes
  Domain: Type
  Range: Type
```

Listing 3.49: The software ontology (soft): types.

Pointers

A `Pointer` is defined as a special kind of variable: it `pointsTo` a variable, which means that it has the address of the memory location of that variable as its value. We also defined the relationship `pointsToType` as a chain of the `pointsTo` and `hasType` relationships, meaning that `A pointsTo B` and `B hasType C` if and only if `A pointsToType C`. We added this relationship to describe pointers which may point only to variables that have a certain type, but which do not point to a specific variable (yet). See listing 3.50.

```
Class: Pointer
  EquivalentTo: Variable and pointsTo some Variable
  SubClassOf: pointsToType some Type
  SpinRule:
    CONSTRUCT {
      ?this expr:hasValue ?address
    } WHERE {
      ?this :pointsTo ?variable .
      ?variable :represents/:hasAddress ?address
```

```
84      }
85
86    ObjectProperty: pointsTo
87      Domain: Pointer
88      Range: Variable
89
90    ObjectProperty: pointsToType
91      Domain: Pointer
92      Range: Type
93      SubPropertyChain: pointsTo o hasType
```

Listing 3.50: The software ontology (soft): pointers.

Namespaces and libraries

Namespaces and libraries are `Container`s according to our *software* ontology: see listing 3.51. Namespaces are elements of the language, while libraries are documents with a file name.

```
94    Class: Namespace
95      SubClassOf: cont:contains only LanguageElement
96      SubClassOf: LanguageElement
97
98    Class: Library
99      SubClassOf: cont:contains only LanguageElement
100     SubClassOf: doc:Document
101     SubClassOf: doc:hasFileName some xsd:string
```

Listing 3.51: The software ontology (soft): namespaces and libraries.

Interfaces

Software interfaces were introduced earlier by the *electricity* ontology for defining I/O modules (see 3.3.16). They are specializations of the `Interface` class of the *systems* ontology, with the additional restriction that they only have software variables as their properties: see listing 3.52.

```
102   Class: Interface
103     SubClassOf: sys:Interface and sys:hasProperty only Variable
```

Listing 3.52: The software ontology (soft): interfaces.

Calls

A `Call` represents the operation of calling a `Callable`, such as a function, a method, a program, etc. The type of a callable (a `CallableType`) consists of an `Implementation` and, possibly, some arguments and a return value. We

constrained the meaning of `Implementation` by saying that it is an ordered container (a `cont:List`) of expressions: see listing 3.53.

```
Class Call:
  SubClassOf: Operation
  EquivalentTo: calls some Callable

ObjectProperty: calls
  SubPropertyOf: hasOperand
  Domain: Call
  Range: Callable

Class: Callable
  SubClassOf: Variable
  SubClassOf: hasType some CallableType

Class: CallableType
  SubClassOf: Type
  SubClassOf: hasImplementation min 1

Class: Implementation
  SubClassOf: cont:List and cont:contains only Expression

ObjectProperty: hasImplementation
  SubPropertyOf: sys:hasPart
  Domain: CallableType
  Range: Implementation

ObjectProperty: hasReturnType
  Domain: Callable
  Range: Type

Class: Argument
  EquivalentTo: Variable and sys:isPropertyOf some CallableType
```

Listing 3.53: The software ontology (soft): calls.

Enumerations

An `Enumeration` in software is a ordered list of enumeration items: see listing 3.54.

```
Class: Enumeration
  SubClassOf: cont:List
  SubClassOf: cont:contains only EnumerationItem

Class: EnumerationItem
  SubClassOf: Variable
  EquivalentTo: cont:isContainedBy exactly 1 Enumeration
```

Listing 3.54: The software ontology (soft): enumerations.

If-then

Our *software* ontology also defines the if-then construct, because it may be reused by many programming languages. See listing 3.55 for the formal definition of the `IfThen` operation and of the `if`, `then`, and `else` properties.

```
142  Class: IfThen
143    SubClassOf: Operation, LanguageElement
144    EquivalentTo: (if some expr:Expression) and (then some Implementation)
145
146  ObjectProperty: if
147    SubPropertyOf: hasOperand
148    Domain: IfThen
149    Range: expr:Expression
150
151  ObjectProperty: then
152    SubPropertyOf: hasOperand
153    Domain: IfThen
154    Range: expr:Implementation
155
156  ObjectProperty: else
157    SubPropertyOf: hasOperand
158    Domain: IfThen
159    Range: expr:Implementation
```

Listing 3.55: The software ontology (soft): if-then.

Qualifiers

Finally, our *software* ontology defines `qualifiers`: additional properties (metadata) about variables. Qualifiers may be used, for instance, to mark variables as read-only and exposed by a data communication server.

```
160  Class: Qualifier
161    SubClassOf: sys:isPropertyOf some Variable
162    SubClassOf: expr:hasValue exactly 1
```

Listing 3.56: The software ontology (soft): qualifiers.

3.3.18 IEC 61131

The *IEC 61131* ontology is the most specific ontology that we created: it represents some of the concepts of the IEC 61131 standard for programmable logic controllers (PLCs). Only the concepts of *Part 3 – Programming languages* (IEC 61131-3) are currently represented. As will be seen below, many of these concepts can be related to the concepts defined by more abstract ontologies such as our *software* ontology.

The starting point of the ontology is the `iec61131` individual, which represents the software language specified by the standard: see listing 3.57. Although IEC 61131-3 specifies several syntaxes (such as structured text (ST), function block diagram (FBD), ladder diagram (LD), etc.), our ontology considers it as one language since only the primitives are represented – and not the syntax. The class `IEC61131Element` represents the elements of this language. We also defined the data property `hasSymbol` to assign IEC 61131-specific symbols to reusable concepts of other ontologies.

```
Ontology <http://www.mercator.iac.es/onto/metamodels/iec61131>
  Import: <http://www.mercator.iac.es/onto/metamodels/software>

Individual: iec61131
  Types: soft:Language

Class: IEC61131Element
  SubClassOf: soft:LanguageElement
  Facts: sys:isElementOf value iec61131

DataProperty: hasSymbol
  SubPropertyOf: soft:hasSymbol
```

Listing 3.57: The IEC 61131 ontology (iec61131).

Operations

Several operations of the IEC 61131 standard have the same semantics as those defined by our *expressions* ontology. Hence, we can reuse those operations, and assign IEC 61131-specific symbols to them. Listing 3.58 also shows the definition of the unary dereference operator, which we could not reuse from other ontologies.

```
expr:and                     hasSymbol  "AND"^^xsd:string
expr:not                     hasSymbol  "NOT"^^xsd:string
expr:or                      hasSymbol  "OR"^^xsd:string
expr:isAssignedTo            hasSymbol  ":="^^xsd:string
expr:isGreaterThan           hasSymbol  ">"^^xsd:string
expr:isGreaterThanOrEqualTo  hasSymbol  ">="^^xsd:string
expr:isLessThan              hasSymbol  "<"^^xsd:string
expr:isLessThanOrEqualTo     hasSymbol  "<="^^xsd:string
expr:plus                    hasSymbol  "+"^^xsd:string
expr:minus                   hasSymbol  "-"^^xsd:string
expr:times                   hasSymbol  "*"^^xsd:string
expr:dividedBy               hasSymbol  "/"^^xsd:string
expr:unaryMinus              hasSymbol  "-"^^xsd:string
expr:absOf                   hasSymbol  "ABS"^^xsd:string

Individual: dereference
  Types: expr:UnaryOperator
  Annotations: hasSymbol "^"^^xsd:string
```

Listing 3.58: The IEC 61131 ontology (iec61131): operations.

Data types

Our *IEC 61131* ontology defines the following data types: `BOOL`, `BYTE`, `DINT`, `DWORD`, `INT`, `LINT`, `LREAL`, `REAL`, `SINT`, `STRING`, `UDINT`, `UINT`, `ULINT`, `USINT`, and `WORD`. Using the `owl:sameAs` relationship, we were able to express that some of them are synomyms of the data types that were defined earlier by the *expressions* ontology. It means that when modeling concrete systems using Ontoscript, we can use the data types of the *expressions* and *IEC 61131* ontologies interchangeably. Other data types such as `WORD` and `DWORD` are specific to IEC 61131, and can therefore only be expressed via the *IEC 61131* ontology. Listing 3.59 only shows a small excerpt of the data type definitions, for the conciseness of this text.

```
Class: DataType
  EquivalentTo: DataType and IEC61131Element

Individual: BOOL
  Types: DataType
  SameAs: expr:t_bool
  Facts: hasSymbol "BOOL"

Individual: USINT
  Types: DataType
  SameAs: expr:t_uint8
  Facts: hasSymbol "USINT"

Individual: SINT
  Types: DataType
  SameAs: expr:t_int8
  Facts: hasSymbol "SINT"

# ... and so on
```

Listing 3.59: The IEC 61131 ontology (iec61131): data types.

Enumerations, structures, and POUs

IEC 61131-3 defines several types of variables: enumerations, structures, and three "POUs" or "program organization units" (functions, function blocks, and programs). They can be easily related to more abstract concepts of the `software` ontology, as shown in listing 3.60.

```
Class: Enum
  EquivalentTo: soft:Enumeration and IEC61131Element

Class: Struct
  SubClassOf: soft:Type
  SubClassOf: soft:hasImplementation max 0

Class: ProgramOrganizationUnit
  SubClassOf: soft:CallableType
  SubClassOf: IEC61131Element

Class: Function
```

```
62    SubClassOf: ProgramOrganizationUnit
63
64  Class: FunctionBlock
65    SubClassOf: ProgramOrganizationUnit
66
67  Class: Program
68    SubClassOf: ProgramOrganizationUnit
```

Listing 3.60: The IEC 61131 ontology (iec61131): enumerations, structures, POUs.

POU attributes and methods

IEC 61131 defines several types of members of POUs: input variables, output variables, input-output variables (similar to *references* in C++), and methods (a recent addition to the standard). These types of variables are poorly constrained by our ontology: for instance, according to our ontology, there is no formal difference between input and output variables. Naturally however, our ontology does not *need* to define these concepts precisely. In fact, the precise definition of these concepts is the sole responsibility of the IEC 61131 specifications, which means that any statement that our ontology makes increases the risk of over-constraining the concepts – and thereby violating the standard specifications.

```
69  Class: InputVariable
70    SubClassOf: soft:Variable
71    SubClassOf: IEC61131Element
72    SubClassOf: isInputVariableOf some ProgramOrganisationUnit
73    DisjointWith: OutputVariable, InOutVariable, Method
74
75  ObjectProperty: hasInputVariable
76    Domain: ProgramOrganizationUnit
77
78  ObjectProperty: isInputVariableOf
79    InverseOf: hasInputVariable
80
81  # ... similar for:
82  #      OutputVariable / hasOutputVariable / isOutputVariableOf
83  #      InOutVariable / hasInOutVariable / isInOutVariableOf
84  #      LocalVariable / hasLocalVariable / isLocalVariableOf
85
86  Class: Method
87    SubClassOf: soft:CallableType
88    SubClassOf: IEC61131Element
89    SubClassOf: soft:isVariableOf some FunctionBlock
```

Listing 3.61: The IEC 61131 ontology (iec61131): POU attributes and methods.

3.4 Models: system models

In the previous section we focused on the metamodels, which represent domain concepts such as *systems*, *requirements*, *power supplies*, *IEC 61131-3 function*

blocks, etc. In this section, we will elaborate how these domain concepts can be used to *specify* models that represent particular systems, such as the electronics and software of the Mercator telescope control system.

3.4.1 Ontoscript

As explained in 2.2.2, we need to add a facade over the ontologies of the previous section, via a set of clear, concise, textual DSLs. According to our framework, a DSL only provides i) a rigorous syntax, and ii) a mapping between language primitives and the terms of the corresponding ontology. As a result, any DSL expression can be translated into a set of expressions in the knowledge representation language of the corresponding ontology. For this purpose, we developed a software library called *Ontoscript*[9].

Ontoscript provides a small DSL with a few primitives such as `METAMODEL`, `MODEL`, `REQUIRE`, `READ`, `WRITE`, ..., to create and populate metamodels and models. For each ontology we created a corresponding Ontoscript file. For instance, listing 3.62 shows a small excerpt of the Ontoscript file `development.coffee`, which corresponds to the *development* ontology. When executed, this Ontoscript file first constructs a `METAMODEL` object called `dev` in the global namespace, and then populates it by reading the *development* ontology. As the ontology is being read (using the `READ` instruction), each class, property or individual of the ontology is added as a new member of the `dev` metamodel. For instance, when the OWL class `dev:Requirement` is read, then a new member called `Requirement` is added to the `dev` metamodel. Since `dev` sits in the global namespace, the OWL class `dev:Requirement` is thus mapped to `dev.Requirement`. Optionally, we can use the `REQUIRE` primitive to load other metamodels in the global namespace. Hence, in case of the shown example, loading the `dev` metamodel will automatically result in loading the `cont` (containers), `fsm` (finite state machines), `mod` (models), and `org` (organizations) metamodels. These metamodels, in turn, will load their own dependencies such as the `expr` (expressions) metamodel, and so on.

```
require "ontoscript"

REQUIRE "metamodels/containers.coffee"
REQUIRE "metamodels/finitestatemachines.coffee"
REQUIRE "metamodels/models.coffee"
REQUIRE "metamodels/organizations.coffee"

METAMODEL "http://www.mercator.iac.es/onto/metamodels/development" : "dev"

dev.READ "metamodels/development.jsonld"
```

Listing 3.62: The `dev` metamodel (excerpt of development.coffee).

We can now create a model of a particular system. For instance, if we want to create a very simple conceptual model of the tertiary mirror (M3) of the

[9]Online at `https://github.com/IvS-KULeuven/ontoscript`.

Mercator telescope, then we can write a script as shown in listing 3.63. This script performs the following actions:

- Line 1: the ontoscript primitives `REQUIRE`, `METAMODEL`, etc. are added to the global namespace.
- Line 3: the required metamodel `dev` is added to the global namespace. This, in turn, adds the metamodels `cont`, `fsm`, `mod`, `expr`, and so on.
- Lines 7-27: several individuals are constructed and related to each other.
- Line 30: the `WRITE` instruction will generate an OWL ontology with IRI `http://www.mercator.iac.es/onto/models/m3` and with prefix `m3`, serialized in json-ld format. During the `WRITE` instruction, the newly created Ontoscript individuals will be mapped to OWL individuals (e.g. `m3.concept` will be mapped to `m3:concept`, `m3.concept.rFoc` will be mapped to `m3:concept.rFoc`, and so on).

```
1  require "ontoscript"
2
3  REQUIRE "metamodels/development.coffee"
4
5  MODEL "http://www.mercator.iac.es/onto/models/m3" : "m3"
6
7  m3.ADD dev.Concept "concept" : [
8    sys.hasProperty dev.Requirement "rFoc" : [
9      rdfs.comment "M3 shall be able to direct the beam to all focal
           stations"
10   ]
11   sys.hasProperty dev.Requirement "rCas" : [
12     rdfs.comment "M3 shall be able to direct the beam to Cassegrain"
13     dev.isDerivedFrom $.rFoc
14   ]
15   sys.hasProperty dev.Requirement "rNasA" : [
16     rdfs.comment "M3 shall be able to direct the beam to Nasmyth A"
17     dev.isDerivedFrom $.rFoc
18   ]
19   sys.hasProperty dev.Requirement "rNasB" : [
20     rdfs.comment "M3 shall be able to direct the beam to Nasmyth B"
21     dev.isDerivedFrom $.rFoc
22   ]
23   fsm.hasStatus fsm.Status "beamStatus" : [
24     cont.contains fsm.State "cassegrain"
25     cont.contains fsm.State "nasmythA"
26     cont.contains fsm.State "nasmythB"
27   ]
28 ]
29
30 m3.WRITE "models/mtcs/m3.jsonld"
```

Listing 3.63: Example of a basic Ontoscript model.

Ontoscript also defines a reference to the execution context via the `self` keyword (very similar to `self` in Python or `this` in C++). In case of nested individuals, we can refer to the "upper" execution context via the `$` keyword (for instance, in the above example, `$.rFoc` stands for `m3.concept.rFoc`).

The `$` keyword can be used recursively to go up multiple levels in the hierarchy (e.g. to refer to the `m3.concept.rFoc` requirement from within the `m3.concept.beamStatus.cassegrain` execution context, we have to write `$.$.rFoc`).

To simplify the creation of individuals, Ontoscript supports the definition of "macros": instructions that transform some input sequence to an output sequence. For instance, listing 3.64 shows the definition of the macros `m3.R_FUNCTIONAL` and `m3.R_BEAM`, which transform some parameters into `Requirement` individuals. The code of listing 3.64 can thus be expanded to lines 7-22 of the previous listing (3.63).

```
1  m3.ADD dev.Requirement "R_FUNCTIONAL" : (args) -> [
2    rdfs.comment "M3 shall be able to #{args.function}"
3  ]
4
5  m3.ADD dev.Requirement "R_BEAM" : (args) -> [
6    APPLY m3.R_FUNCTIONAL(function: "direct the beam to #{args.name}")
7    dev.isDerivedFrom args.derived
8  ]
9
10 m3.ADD dev.Concept "concept" : [
11
12   sys.hasProperty m3.R_FUNCTIONAL(function: "direct the beam to all focal
         stations") "rFoc"
13
14   sys.hasProperty m3.R_BEAM(name: "Cassegrain", derived: $.rFoc) "rCas"
15   sys.hasProperty m3.R_BEAM(name: "Nasmyth A" , derived: $.rFoc) "rNasA"
16   sys.hasProperty m3.R_BEAM(name: "Nasmyth B" , derived: $.rFoc) "rNasB"
17   # ...
```

Listing 3.64: Example of a model, using macros.

Without support for macros, Ontoscript would offer very few advantages over existing Semantic Web syntaxes such as Turtle. With support for macros however, we can create new "frame-based" DSLs at the modeling level. For instance, lines 14-16 of listing 3.64 illustrate the use of the "frame" `R_BEAM` with slots `name` and `derived`.

Aside from the support for macros and support for the `self` and `$` keywords, Ontoscript is very different from existing Semantic Web languages because it is an *internal* domain-specific language. An internal (or *embedded*) DSL is written in an executable "host" language and is parsed by executing the DSL within that language [64]. The DSL only uses a restricted set of the host language's features, in a particular style. Conversely, *external* DSLs are parsed by an external program. External DSLs have the advantage that they are not restricted by the syntactic rules of a host language, so their syntax can be crafted much more freely.

The host language of Ontoscript is CoffeeScript: a programming language that can be translated (or "transcompiled" or "transpiled") one-to-one into JavaScript. According to the CoffeeScript website[10], "coffeescript is an attempt

[10] `http://coffeescript.org`

to expose the good parts of JavaScript in a simple way". CoffeeScript is very succinct: it removes delimiters such as curly brackets ({}) in favor of white space (much like Python does), and introduces some new features and "syntactic sugar" to improve readability and productivity [61]. It means that all previously shown Ontoscript code is valid CoffeeScript code, used in a particular style. For instance, the DSL primitive `MODEL` is in reality a CoffeeScript function with an CoffeeScript Object as its argument: see listing 3.65.

```
1  MODEL "http://www.mercator.iac.es/onto/models/m3" : "m3"
2  # is equivalent to:
3  MODEL( { "http://www.mercator.iac.es/onto/models/m3" : "m3" } )
```

Listing 3.65: Example of CoffeeScript's syntactic sugar.

Despite having a number of "bad parts" (some of which are listed in chapter 7 of [61]), CoffeeScript is very suitable for serving as the host language of DSLs due to its readability and its support for (nested) *closures* and *mixins*. Macros in Ontoscript are examples of CoffeeScript closures: for instance, the CoffeeScript function `R_BEAM` returns another CoffeeScript function (instead of a value), which is called in listing 3.64 with simple string arguments. The `APPLY` instruction at line 6 of the same code listing is an example of a mixin: it "mixes in" the facts produced by the `R_FUNCTIONAL` call as if these facts are produced by the `R_BEAM` call.

An advantage of using an internal DSL is that we have the "expressive power" of the host language (CoffeeScript in this case) at our disposal. It means that, aside from the ontologies described in the previous section, we have an additional "metamodel" which provides concepts such as *loops*, *conditionals*, *arrays*, etc. As we experienced, the additional expressiveness provided by CoffeeScript turned out to be very helpful to efficiently model particular systems, which often have a lot of potential for reuse. As a very simple example, we could replace lines 15-16 of listing 3.64 by the *for* loop of listing 3.66. Obviously this example may be too simple to justify the use of the *for* loop, but in many other cases (such as the modeling of 25 pins of a connector) it is very helpful.

```
1    for n in ['A', 'B']
2      sys.hasProperty m3.R_BEAM(name    : "Nasmyth #{n}",
3                                derived : $.rFoc        ) "rNas#{n}"
```

Listing 3.66: Example of using CoffeeScript's expressive power in Ontoscript.

The expressiveness of CoffeeScript also allows us to create more advanced macros. For instance, listing 3.67 shows how macros are created and added to a `factories` model – a model designed to be reused by application-specific models such as our `m3` model. The last line of this code listing creates a new

CoffeeScript function called `MAKE_CONCEPT` and adds it the global namespace (`root`).

```
1  factories.ADD dev.Requirement "FUNC_REQ" : (args) -> [
2    if args.function?
3        rdfs.comment "The system shall be able to #{args.function}"
4    if args.derived?
5        dev.isDerivedFrom args.derived
6  ]
7
8  factories.ADD dev.Concept "CONCEPT" : (args) -> [
9    if args.functional_reqs?
10     for name, details of args.functional_reqs
11       sys.hasProperty common.FUNC_REQ(details()) name
12 ]
13
14 root.MAKE_CONCEPT = (name, args) -> factories.CONCEPT(args) name
```

Listing 3.67: Example of more advanced Ontoscript macros.

Using the `MAKE_CONCEPT` function, we can now create very "clean" models, as illustrated in listing 3.68. Effectively, we have created a new DSL, with a primitive called `MAKE_CONCEPT`.

```
1  m3.ADD MAKE_CONCEPT "concept",
2    functional_reqs:
3       rFoc  : -> function: "direct the beam to all focal stations"
4       rNasA : -> function: "direct the beam to Nasmyth A", derived: $.rFoc
5       rNasB : -> function: "direct the beam to Nasmyth B", derived: $.rFoc
```

Listing 3.68: Usage example of the macros defined in listing 3.67.

We found the ability to "program" a model rather than to simply "draw" a model (as in the case of UML/SysML) very useful to express *realization* (sometimes referred to as "instance modeling"). For example, consider the semantics of the `hasType` relationship of the *software* ontology (see 3.3.17). The `hasType` relationship is a sub-property of the `realizes` relationship of the *systems* ontology (see 3.3.1). It means that if a software variable `m1Temp` has function block `FB_Temperature` as its type, then all attributes (and sub-attributes) of `FB_Temperature` must be realized by `m1Temp`: see listing 3.69. The `soft.VARIABLE` macro call of line 17 therefore inspects the type, recursively creates new "instance attributes" for all "type attributes", and creates `sys:realizes` relations between corresponding attributes. So when line 17 is executed, then a complex individual `m1Temp` is created, consisting of attributes and sub-attributes. In the next lines, we can now express new facts about these newly created attributes, as shown in line 19. Furthermore, the complex individual `m1Temp` will be consistent according the semantic rules of the *systems* ontology, because all parts of the realized system `FB_Temperature` are now indeed realized by the parts of `m1Temp`. Some SysML/UML CASE tools provide similar functionality via plug-ins or special procedures (such as

the "Automatic instantiation" wizard of the MagicDraw tool[11]), which we found much less convenient since they require manual interaction with the CASE tool for every "instantiation" (or *realization*, as we call it).

```
commonsoft.ADD MAKE_FUNCTION_BLOCK "FB_Temperature",
  var_in:
    rawValue   : -> type: t_int16 , comment: "The raw measured value"
    conversion : -> type: t_double, comment: "DegC=rawValue*conversion"
  var_out:
    celsius    : -> type: commonsoft.QuantityValue
    kelvin     : -> type: commonsoft.QuantityValue
  calls:
    celsius:
      value    : -> MUL(self.rawValue, self.conversion)
      unit     : -> commonsoft.Units.DEGREES_CELSIUS
    kelvin:
      value    : -> SUM(self.celsius.value, DEGREES_CELSIUS_TO_KELVIN)
      unit     : -> commonsoft.Units.KELVIN

# add variable "m1Temp" of type FB_Temperature to the telemetry subsystem:
telemetry.ADD soft.VARIABLE(type: commonsoft.FB_Temperature) "m1Temp"

telemetry.m1Temp.celsius.value.ADD soft.hasQualifier opcua.READ_ONLY
#               ^^^^^^^^^^^^^^^
#               now we can refer to all (sub)attributes of m1Temp!
```

Listing 3.69: Representative usage example of Ontoscript.

Several more examples of Ontoscript models that we developed for the Mercator TCS are displayed in appendix A.

3.5 Tooling: OntoManager

To take advantage of the reusability of our ontologies and models, we have developed a tool called OntoManager[12]. OntoManager can be broken down into several related parts; a (graphical, informal) architectural model is shown in figure 3.4.

The central part of OntoManager is the knowledge base: a graph store which stores both the metamodels and models of our framework. OntoManager also has a rules engine which supports OWL 2 RL, OWL, and SPIN, to infer *implicit* facts based on the existing *asserted* facts of the metamodels and models. When the rules engine is executed, this implicit knowledge is added to the knowledge base. OntoManager also has a template engine, which can generate "views" of the models – as HTML web pages to be read by humans, and as source code to be compiled by external software. When a user requests a particular view using his/her web browser, then the corresponding template is loaded, the necessary queries are executed to populate the template, and the result is sent back to the web browser of the user.

[11] Company website: `https://www.nomagic.com`.
[12] Open-source and online at `https://github.com/IvS-KULeuven/OntoManager`.

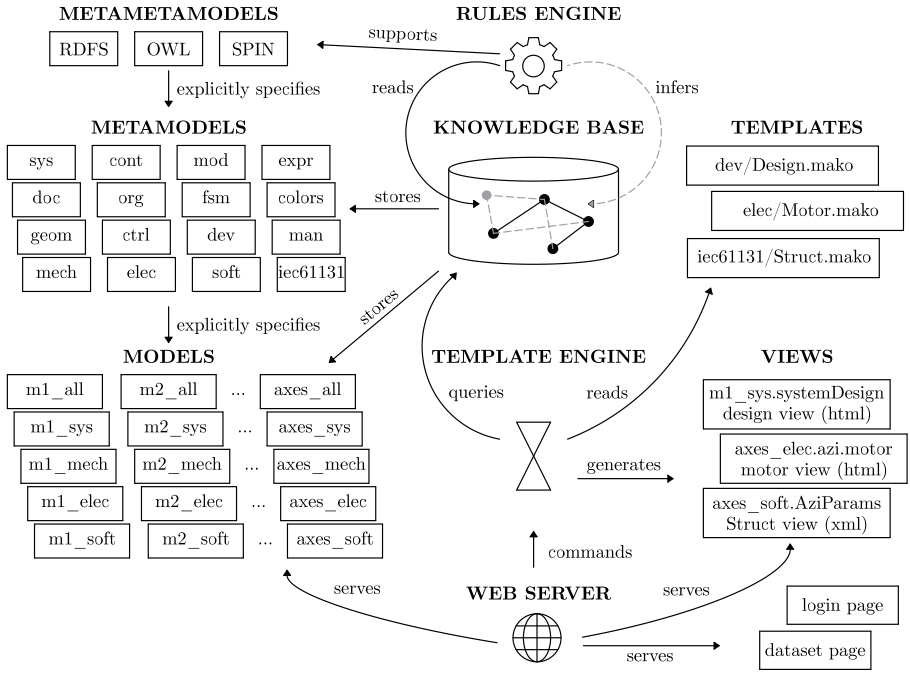


Figure 3.4: Architectural model of OntoManager.

In the remainder of this section, we will explain the parts of OntoManager that were not discussed before.

3.5.1 Web server

OntoManager has a built-in web server, allowing users to access all functionality using their web browser. Several screen captures of OntoManager, as seen via a web browser, are depicted in appendix B. The web server is based on the open source Pyramid web framework[13], which provides a collection of Python packages to build web servers based on the popular model-view-controller (MVC) design pattern.

OntoManager is a multi-user application: users must first enter their name and password via a **Login page**, when first accessing the URL of the web server. If valid credentials are entered, then a cookie-based session is created, and the user is directed to his/her **Home page**. The latter is very simple in the current implementation: it only shows the "groups" to which the user belongs, and download links to the contents of the "home directory" of the user. The groups determine the access rights of the user: e.g. some users have read-only access to the knowledge base (and thus only have the ability to request "views" of the models), while other users can also manipulate the knowledge

[13]See `http://www.pylonsproject.org`.

base by commanding the inference engine, by entering free queries, and so on. The ability of a user to download files from a "home directory" was added to communicate large documents with particular end-users in a convenient way.

All users have access to the **Models page**, which allows them to navigate to a particular Ontoscript model, and display this model in the web browser. In the current implementation the models can only be read, but future versions of OntoManager should foresee write access (and built-in version control capabilities) – so that models can be created and edited only using OntoManager, without the need for an external editor.

Some users may also access the **Dataset page**, depending on the groups they belong to. Using this web page, one can tick the following checkboxes and start a process on the web server host machine:

- *run metamodels*: convert the ontologies from Turtle to json-ld format;
- *run models*: execute the selected Ontoscript files (on the command line, via Node.js[14]), thereby converting the Ontoscript models into json-ld named graphs;
- *run inferences*: start the SPIN API rules engine;
- *load asserted data*: feed the asserted facts (those explicitly described by the models) into the knowledge base;
- *load implicit data*: feed the implicit facts (those generated by the inference engine) into the knowledge base;
- *generate PLCopen files*: generate the selected PLC source code files;
- *save cache*: store the current query results in a file, allowing OntoManager to be restarted without the need to re-execute all queries.

The **Problems page** lists the problems that were produced by the constraint rules (`SpinConstraint`) of the ontologies. As discussed earlier in this chapter, SPIN constraints are evaluated in the closed world: they produce warnings and errors (as new individuals of type `spin:ConstraintViolation`) if facts are missing or inconsistent.

The **Browse page** simply lists all triples whose subject is a given RDF resource. Using this page, one can navigate from one resource to another, using the semantic relations between them.

Users can enter "free" SPARQL queries, using the **Query page**. Access to this page is restricted, since it can be used to manipulate the knowledge base (e.g. one can add new relations and nodes to the knowledge base via the `CONSTRUCT {} WHERE {}` clause of SPARQL). An example of such a free query is shown in appendix figure B.7.

[14] Node.js® is a JavaScript runtime platform built on the V8 JavaScript engine of the Chrome web browser by Google: see `https://nodejs.org`.

The remaining pages of OntoManager are dynamically generated: they represents views of the models that are useful to systems engineering (the **Systems page**), mechanical engineering (the **Mechanics** page), electronics (the **Electronics** page), and software engineering (the **Software** page).

3.5.2 Rules engine

As already mentioned in 3.2.3, we use the open-source SPIN API software to process the semantic rules of the metamodel ontologies. SPIN API builds further on Apache Jena[15], by adding support for the SPIN vocabulary.

Rules are executed in two stages. In the first stage, all SPIN rules are executed cyclically, including those of the OWL 2 RL profile. Every cycle produces new facts (the so-called *inferred* or *implicit* knowledge), and adds these facts to the knowledge base. This cyclic process continues until no new facts can be generated. At that point, the `SpinConstraint` rules can be executed in a single cycle, to produce a `spin:ConstraintViolation` individual for each inconsistency of the models. By adding these individuals to the knowledge base, OntoManager can query for them, and generate the **Problems page**.

3.5.3 Knowledge base

In the current implementation, OntoManager uses RDFLib to store the triples in a so-called "conjunctive graph", which represents an unnamed aggregation of named graphs. RDFLib is an open-source Python library[16] for working with RDF data. It provides several ways to store RDF data (e.g. as a conjunctive graph), a SPARQL query endpoint, and parsers and serializers for several formats (including Turtle and json-ld). We chose RDFLib because it is versatile, and because it is written in Python and therefore easy to integrate in OntoManager.

3.5.4 Views, templates, and the template engine

OntoManager can produce *views*, which we define as artifacts that represent system models for a specific purpose. Since we consider *represents* to be a transitive relationship (see 3.3.3), and since system models *represent* the actual systems (see 3.1.2), it follows that the views indirectly *represent* the actual systems. According to the definitions of 3.1.1, a view is a therefore a *model* of a system model, much like "a painting of a painting". Views convey less information than the actual system models, but they serve a different purpose. They are produced by first querying the knowledge base, and then "filling out" templates with the results of those queries. In its current implementation,

[15] Apache Jena is "a free and open source Java framework for building Semantic Web and Linked Data applications": see `http://jena.apache.org`.

[16] See `https://github.com/RDFLib`.

OntoManager produces two kinds of views: web-based documentation, and source code.

Web-based documentation is to be consumed by humans, using their web browser. Appendix B shows several examples of such views, which are all linked to each other via the semantic relations that hold between the models they represent. To simplify the implementation of OntoManager, we have created templates for several classes. When an individual must be represented, OntoManager can send a query to the knowledge base to request all classes of the individual (for instance, the invididual `m3_elec:io.module3` may belong to the classses `elec:IoModule`, `elec:Consumer`, `elec:Device`, `mech:Assembly`, `sys:System`, and `owl:Thing`). Depending on the requested information by the user, OntoManager may load the template that corresponds to `elec:IoModule` (showing a table about the terminals and connections of the I/O module) or, for instance, the template that corresponds to `owl:Thing` (showing a list of all known facts about the individual). The templates are expressed in the language defined by the Mako[17] library. Once a view is populated by query results, it is stored in the memory of the OntoManager application. The next time the same view is requested, it can therefore be shown almost almost instantaneous. As discussed in 3.5.1, this "cache" memory can be stored persistently, to allow the state of OntoManager to be restored quickly after a restart.

Other templates of OntoManager are very similar, only they produce source code files, to be "consumed" by a compiler. For instance, we have created templates to produce IEC 61131-3 source code, in the standard PLCopen XML format. A user can simply navigate (via the **Software page**) to a software library, and click on a button to generate the XML source code file. By clicking the "download" button, the user can download the generated file to his/her local computer. Due to the web-based nature of OntoManager, the source code file can be transferred very easily to the computer that is used to compile the code. Other templates are available to produce Python code, by querying the same IEC 61131-3 software models (see next chapter).

3.6 Summary

In this chapter, we first modeled the architecture of our framework. By defining terms such as *system*, *model*, *metamodel*, *metametamodel*, *to represent*, and *to specify*, we were able to describe the main components of the framework, and the relations that hold between them. We argued that the metametamodels are knowledge representation languages according to our framework, and that those of the Semantic Web community (more specifically: RDF Schema, OWL 2 RL, and SPIN) are reasonable choices to satisfy our requirements. The chosen knowledge representation languages specify the metamodels of our framework: from the very abstract *systems* ontology to the very specific *iec61131* ontology.

[17] Mako is an open-source template library written in Python. It is very powerful as it allows Python code to be embedded in the templates: see `http://www.makotemplates.org`.

These ontologies specify the models of the actual systems: they provide the semantics of the DSLs that we use to describe the systems. The syntax of these DSLs is provided by our Ontoscript library, which is an internal DSL based on CoffeeScript. We further argued that the expressiveness of Coffeescript was very helpful to model the actual systems efficiently, since it allowed us to "program" the models rather than simply "draw" the models. Finally, we discussed the architecture and the capabilities of OntoManager, our in-house developed tool to "manage" the framework. Using OntoManager, a user can inspect the models, infer the implicit knowledge, perform queries on the knowledge base, and produce views – using nothing more than a web browser.

Chapter 4

Application

HAVING specified and implemented the requirements of the framework in chapter 2 and 3, we can now *apply* the framework to a real-world system: the Mercator Telescope. In this chapter, we will briefly describe the background of this system, the overall control system architecture, and we will zoom into the most important subsystems of the telescope. The focus of this chapter is thus on the technical details of the Mercator Telescope control system, to gain insight into the capabilities and limitations of the current framework implementation. This insight will serve as input for the next chapter (chapter 5 – *Evaluation*).

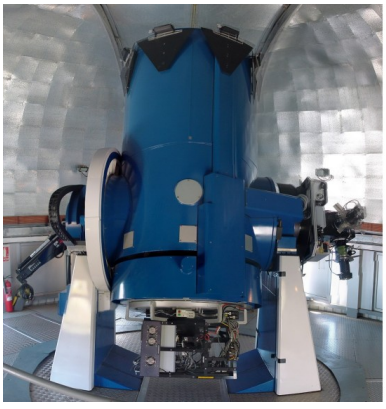
4.1 The Mercator Telescope

The Mercator Telescope is an optical telescope with a primary mirror of 1.2 m in diameter, installed at an altitude of 2333 m above sea level, at the Roque de los Muchachos observatory (La Palma, Canary Islands, Spain). The telescope was constructed in 2000, and has been in operation since 2001 to obtain nighttime science observations, for 360 nights per year. The telescope is owned and operated by the Institute of Astronomy of the KU Leuven, and was funded by the Flemish Community of Belgium. It is mainly used to study stellar astrophysics in two domains: *asteroseismology* and *binary star physics*. While the former domain studies the internal structure of stars by measuring stellar oscillations (much like earthquakes give us information about the internal structure of the Earth), the latter domain studies the binarity[1] of stars and the impact of this binarity on stellar evolution. Two instruments are installed at the telescope to obtain observations: a spectrograph called HERMES and an imager called MAIA. HERMES (High Efficiency and Resolution Mercator Echelle Spectrograph) uses optical fibers to direct the light of the telescope to the main optics of the instrument inside a temperature controlled room, to form

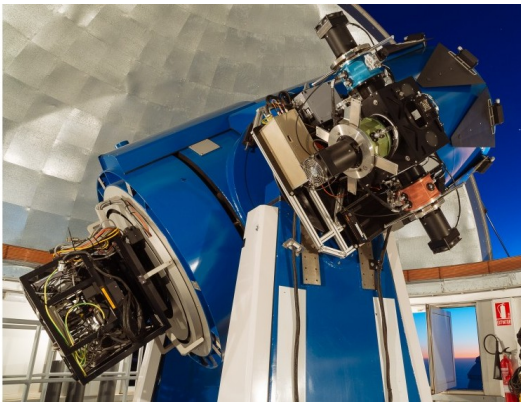
[1] A binary star system consists of two stars that orbit around a common center of mass, and therefore influence each other in various ways.

(a) Roque de los Muchachos Observatory at La Palma.

(b) Mercator Telescope building.



(c) Mercator Telescope.

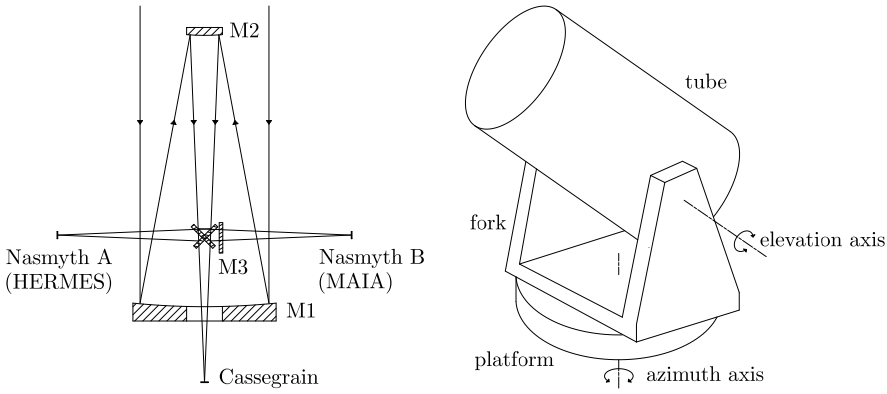


(d) The telescope, ready for observations.

Figure 4.1: The observatory and telescope.
(Picture (d) taken by Péter I. Pápics.)

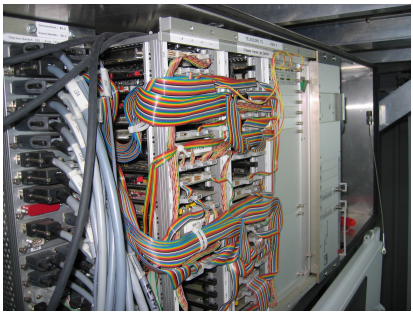
a spectrum from 377 to 900 nm with a high spectral resolution, via a diffraction grating and cross-dispersion prisms [92]. MAIA (Mercator Advanced Imager for Asteroseismology) on the other hand is a three-channel imager: it splits the light beam of the telescope into different wavelength bands (u, g, and $r{+}i$), and uses three science cameras to simultaneously take images of the same field of stars in those wavelength bands [90]. The permanent access to a telescope, equipped with a multi-channel imager and a spectrograph that rivals or outperforms most other world-class instruments in terms of efficiency [92], at a high-quality sky site, has opened a niche to the Institute of Astronomy and its partners. For the Institute of Astronomy, the ability to monitor spectral variability (using HERMES) and brightness variations (using MAIA) over an extended period of time and with high precision, has been essential to continue its leading role in the field of variable single and multiple star research [90]. Figure 4.1 shows some recent pictures of the telescope, instruments, and environment.

The Mercator telescope has a Ritchey-Chrétien design: it uses a concave hyperbolic primary mirror (M1) and a convex hyperbolic secondary mirror (M2) to gather the incoming light of distant objects on the sky: see figure 4.2a. This configuration is widely used by large professional telescopes, as it offers a wide field of view free of optical aberrations. The tertiary mirror (M3) of the Mercator Telescope is a flat mirror that reflects the light coming from M2 outwards of the tube, to one of the instruments. If M3 is moved out of the beam, then the light is passed through a hole of M1 to the so-called *Cassegrain* focal station (where a decommissioned camera called MEROPE [20] is still installed). When M3 is moved inside the beam, then the light is directed through holes of the tube to one of the so-called *Nasmyth* focal stations (where HERMES and MAIA are installed). In summary, the intent of M1 is to gather light from the sky, the intent of M2 is to correct the optical aberrations introduced by M1, and the intent of M3 is to reflect the light out of the telescope to one of the instruments.

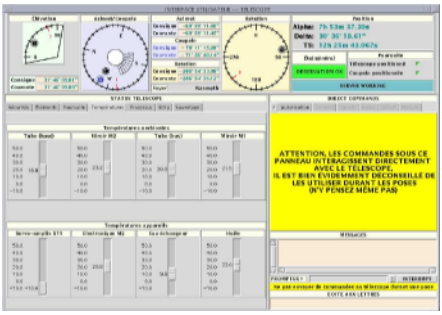


(a) Optical axes. (b) Rotation axes.

Figure 4.2: The telescope's optical and rotation axes.



(a) One of the original transputer racks.



(b) One of the original user interfaces.

Figure 4.3: The original TCS of the Euler and Mercator telescopes.

On a typical night, the Mercator Telescope observes a few dozens of targets on the sky. As the Earth rotates around its rotation axis in about 24 hours, these targets appear to move on the sky around the Pole star (a star visible to the naked eye and which – coincidentally, at this time in history – happens to be located close to the Earth's rotation axis). Thus, for each target, the telescope must first move fast (or *point*) to the target, and then follow (or *track*) the trajectory of the target on the sky at the same (slow) pace. The Mercator Telescope has two axes to perform these movements: an azimuth axis that allows movement parallel to the horizon, and an elevation axis that allows movement between the horizon and zenith (see figure 4.2b). Because the Earth's rotation axis does not coincide with one of the telescope's rotation axes, the beam of light coming from the sky appears to rotate around the optical axis while the azimuth and elevation axes are tracking a target. To counter-act this so-called *field rotation*, two *instrument derotators* are installed at the telescope: one at the Cassegrain and one at the Nasmyth B focal station. By rotating the instruments at the same pace of the field rotation, we can make the observed beam of the sky to "stand still" with respect to the instruments. HERMES at Nasmyth A does not have a derotator, but this is not needed as HERMES only observes the spectrum of a single star at a time through a fiber (regardless of any field rotation).

The Mercator Telescope is a twin of the Leonhard Euler Telescope of the Geneva Observatory, installed at the La Silla Observatory, at the edge of the Atacama desert in Chile. Both telescopes and telescope buildings share the same design and characteristics, but they are located on different hemispheres (thus seeing different parts of the sky), which makes them very complementary. Despite their shared design, a few differences are very noticeable when one visits the telescopes: from the view through the windows (stony desert for Euler, blue sea for Mercator), the paint of the telescope tubes (red for Euler, blue for Mercator), the set of instruments (CORALIE and EulerCam for Euler, HERMES and MAIA for Mercator), ... to the control systems of the telescopes, the subject of this thesis.

Originally these control systems were designed by the Geneva Observatory in the 1990s, using transputer technology – a microprocessor architecture of the 1980s, intended for parallel computing. In the original design, transputers were responsible for "low level" control (interacting directly with the hardware), while a custom software application running on a Solaris system (an Ultra 1 workstation by Sun Microsystems) was responsible for the "high level" control. The transputers and the workstation were connected by SBus: the computer bus by Sun Microsystems before migrating to PCI (Peripheral Component Interconnect). Obsolescence of the aforementioned technologies has forced both the Swiss and Belgian institutes to refurbish the original control system of their telescopes. While the Geneva Observatory opted to migrate the existing software of the Euler Telescope to new platforms (e.g. by "translating" transputer software from Occam to C, by replacing transputers by industrial PCs, and by replacing Sun workstations by Linux computers), the Institute of Astronomy decided to rebuild the whole control system from scratch. This provided them the flexibility to choose other technologies (such as industrial PLCs, I/O, and communication protocols), to improve the performance and functionality of the system, and to gain knowledge and experience by developing everything "in-house". It also gave the institute the opportunity to investigate a new design methodology for telescope control systems, in collaboration with the department of electrical engineering of the KU Leuven (ESAT), which finally resulted in this thesis.

4.2 Control system architecture

The route towards a new Mercator Telescope control system was already taken in 2008, when the control of the dome was moved from the transputers to a Python-based software application running on Linux computers, developed as the subject of my Master's thesis [80]. From this application, a software package called MOCS emerged: the Mercator Observatory Control System. MOCS consists of a typical component-based software framework: it allows rapid development of software components (applications) by providing common technical services such as a communication service (supporting both publisher/subscriber communication and remote procedure calling), a logging service, a configuration service, a component that manages the construction and destruction of other components on predefined host computers, and so on. Between 2008 and 2011, more than 20 components were developed within the MOCS framework. Some of these components were mere proxies of hardware devices (such as I/O modules), others implemented business logic (e.g. to control HERMES, to control the instrument detectors, to control the guiding[2], ...) or graphical user interfaces. The latter typically do not implement any business logic; they are simple front-ends for other components that are running on headless Linux servers. They are

[2] A guiding system corrects the very small position errors of the telescope axes while tracking a target. It continuously takes images of one or more stars through the telescope, measures the drift of the center of these stars with respect to their initial position, and sends the corresponding correction action to the telescope drive system.

Figure 4.4: The user interfaces of MOCS (8 rightmost monitors) and the user interface of the new TCS (leftmost monitor).

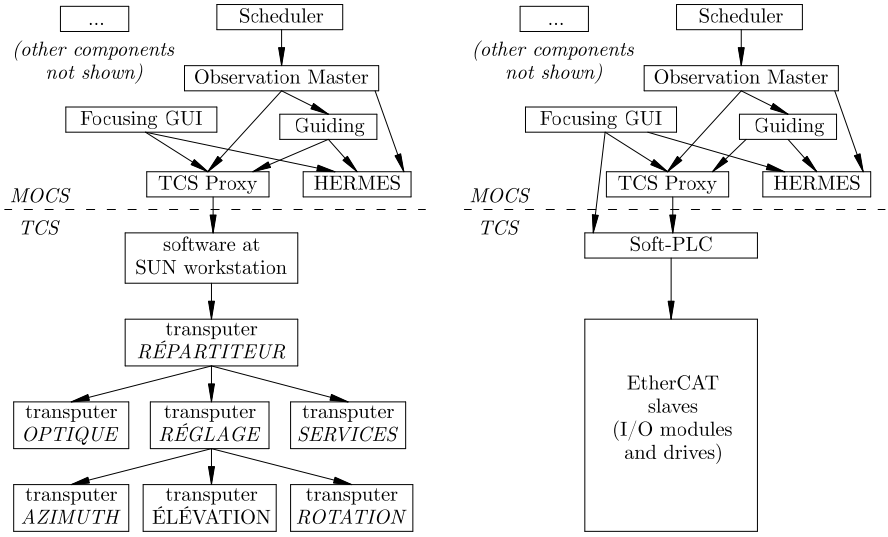
displayed on the 8 rightmost monitors of figure 4.4. Except for some low-level device drivers, all MOCS software has been written in Python.

4.2.1 Overall architecture

The MOCS software package provides all needed "high level" functionality to conduct observations: from semi-automatic scheduling of the targets before the observations, guiding and instrument controlling during the observations, to commanding the data reduction software after the observations. Most of this functionality was already provided by the original TCS, but MOCS components have gradually taken over this functionality, offering many more additional features. As a consequence, the high-level MOCS software has become the "client" of the low-level "server" software of the Telescope Control System (TCS). Informally, the Mercator TCS is responsible for everything which is directly related to the telescope, such as the support and alignment of the mirrors (M1, M2, M3), the control of the hydrostatic bearing of the telescope, the safety system, the axes drive system, and the telescope cover. Section 4.3 will explain these subsystems more in detail.

Figure 4.5 shows a part of the control flow of a typical HERMES observation, before and after the TCS replacement. In the new system, the software running on the SUN workstation has been replaced by a soft-PLC application running on an industrial computer. This soft-PLC application also implements the low-level logic that was previously implemented on the transputers. The fact that all logic is executed on the same device[3] does not necessarily make the new system more "centralized": also the soft-PLC application consists of multiple processes, spread over multiple cores of the CPU. In contrast to the original TCS, the new TCS does not require a proxy anymore: every MOCS component can directly initiate communication with the soft-PLC application, in parallel. This needs to be done carefully however, since parallel access to shared resources opens the door to concurrent computing problems such as race conditions, deadlocks,

[3]Except for the safety logic, which is executed on an EtherCAT slave.



(a) Original TCS in charge of telescope. (b) New TCS in charge of telescope.

Figure 4.5: Control flow from MOCS to the TCS.
(An arrow means: *initiates communication to.*)

and starvation. In the original TCS these problems could not occur, since only sequential access to the TCS was allowed. However, since the control flow of observations is not complex, this "careful" implementation is easy to achieve in practice. The gains of having parallel access are therefore much bigger than the costs, as we can now reduce overhead significantly (for instance, by moving M2, M3 and the axes of the telescope in parallel when pointing the telescope to a new target). While not strictly needed anymore, the original proxy component of MOCS was not removed, but it was adapted to the new TCS, to maintain compatibility with the other MOCS components. As the MOCS components are not modeled in our framework, we cannot simply "factor out" the proxy to increase the evolvability of the overall system.

The technology used to communicate between MOCS and the TCS is OPC[4] Unified Architecture, or OPC UA in short. OPC UA is the platform-independent, secure, modern successor of "classic" OPC, the widely used industrial communication technology of the OPC Foundation. OPC UA is interesting because it defines an expressive modeling language, allowing developers to create "rich" communication interfaces. As we elaborated in the magazine of the OPC Foundation [81], this opens up a wealth of interesting applications that could not be realized by previous widely accepted industrial communication technologies. Unfortunately however, the semantics of the language defined by OPC UA are not formally specified, and are therefore little reusable – much like UML and SysML semantics. Despite this lack of

[4]OPC stands for Openness, Productivity, Collaboration, but used to be an acronym for Object Linking and Embedding (OLE) for Process Control.

formality, the OPC Foundation and its partners have defined several language extensions, to form DSLs. More specifically, a DSL has been defined to express structural models of the IEC 61131-3 PLC programming language. Since our PLC manufacturer supports this DSL, we can:

1. model IEC 61131-3 software, using Ontoscript and our *iec61131* ontology;
2. generate PLCopen XML code from these IEC 61131-3 software models, using OntoManager;
3. import, extend, compile and deploy the generated PLCopen XML code in a commercial PLC programming tool (TwinCAT 3 by Beckhoff);
4. expose the deployed IEC 61131-3 software via the off-the-shelf OPC UA server of the PLC (TwinCAT OPC UA server);
5. generate client-side stubs[5] in Python of the IEC 61131-3 software that we modeled in the first step;
6. use this generated stub code within MOCS to interact very conveniently with the OPC UA server on the PLC.

The support for IEC 61131-3 by our modeling framework and by our PLC and OPC UA server manufacturer, makes it very convenient to create large OPC UA interfaces on the server-side (the PLC) and to interact with those interfaces on the client-side (the MOCS components). A glimpse of the OPC UA interface of the Mercator TCS is shown in figure 4.6. The OPC UA server running on the PLC exposes several thousands of variables in this way, all structured in a hierarchy of 14 levels deep. Naturally, this hierarchy makes the Mercator TCS runtime a very object-oriented, traditional software system. This design was chosen for pragmatic reasons: the telescope *can* be controlled in a hierarchical way, and time constraints did not permit us to extend the "less object-oriented" modeling paradigms of our framework, to the runtime software of the TCS.

Due to the generated stub code, we can access the variables of the OPC UA server using only 3 lines of code, as shown in the example of listing 4.1. The generated code makes extensive use of UAF, the Unified Architecture Framework, an open-source software library which we developed since 2011 (see `https://github.org/uaf/uaf`). Essentially, the UAF takes care of technical concerns that are frequently needed by OPC UA clients. Its philosophy is to provide a "web-browsing" experience to OPC UA users, allowing them to interact directly with OPC UA data without worrying about the connection and security concerns. Originally developed for the Mercator Telescope, our UAF software is now used in many other application areas – from testing I/O boards at CERN (ATLAS experiment), to interfacing PLCs and Human Machine Interfaces (HMIs) of machines for the food industry.

[5] A stub is a piece of code that represents (acts in place of) some functionality.

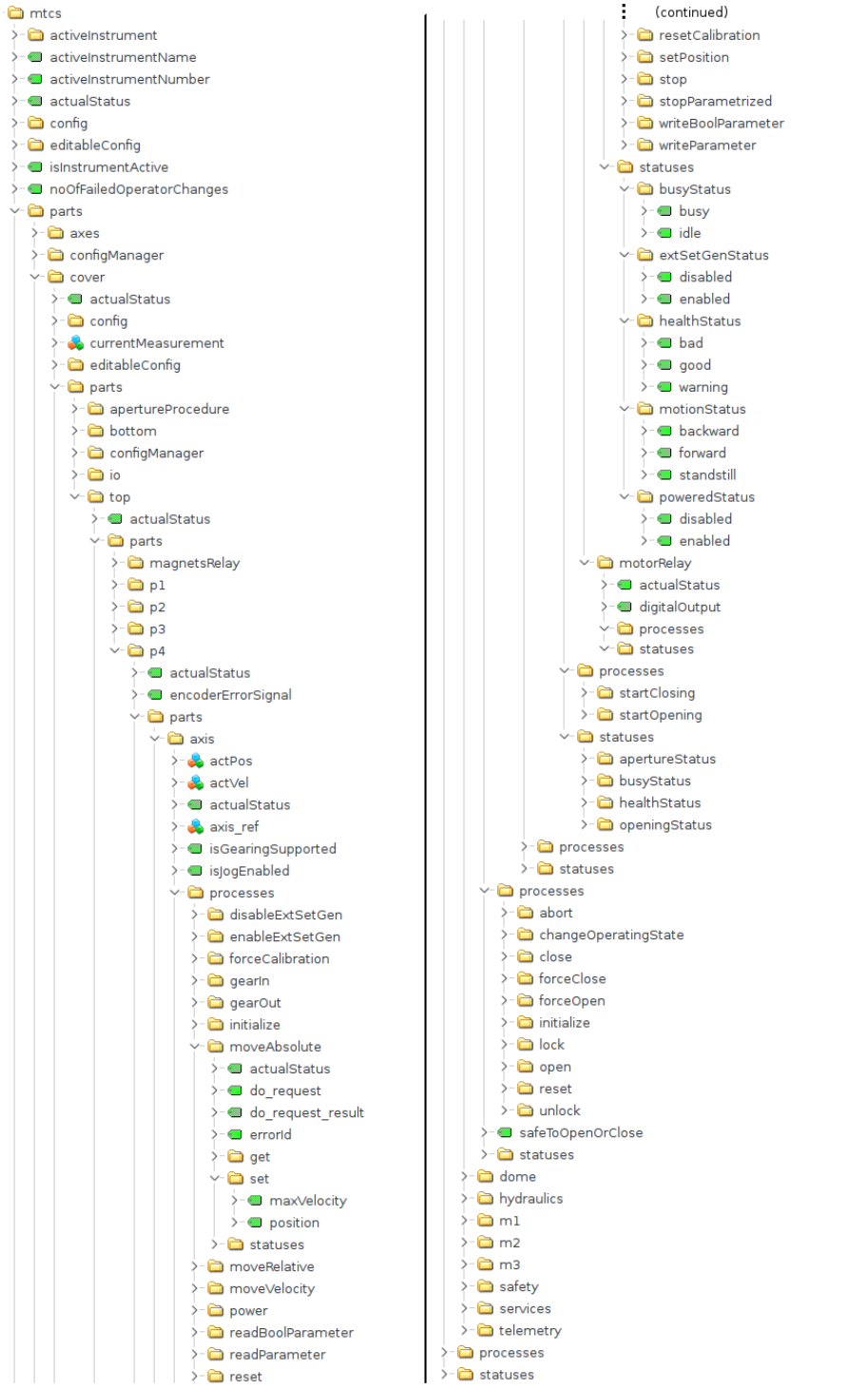


Figure 4.6: A very small excerpt of the OPC UA address space of the PLC.

```
>>> import opcua
>>> c = opcua.buildClient()
>>> c.read(opcua.MTCS.parts.axes.parts.azi.actPos.degrees.value.ADR())
 - overallStatus                : Good
 - requestHandle                : 2
 - targets[]
    - targets[0]
       - clientConnectionId     : 0
       - status                 : Good
       - opcUaStatusCode        : 0
       - data                   : 24.460292997266546
       - sourceTimestamp        : 2016-11-25T11:19:14.717Z
       - serverTimestamp        : 2016-11-25T11:19:14.717Z
       - sourcePicoseconds      : 0
       - serverPicoseconds      : 0
```

Listing 4.1: Example of how generated Python stub code can be used to read TCS variables from within MOCS.

4.2.2 TCS architecture

An overview of the main TCS components, and their communication links, is shown in figure 4.7. Most prominent in this figure is the soft-PLC which runs all control logic of the system – except for the safety logic. Unlike a regular PLC, a soft-PLC is a software product that can run on a variety of devices – from "regular" desktop computers to small embedded devices. Soft-PLCs typically run all control logic in real-time, and a general-purpose operating system with "user-space" applications in non-real-time. *Real-time* means that the soft-PLC schedules the execution of the tasks (processes) at a fixed frequency. For instance, the Mercator TCS soft-PLC schedules its tasks at 1000 Hz. Every millisecond, the soft-PLC determines which tasks must be executed (depending on their cycle time), and in which order (depending on their priority). The selected tasks are then executed sequentially, thereby "occupying" the CPU of the device without interference of other processes. The CPU is "given back" to the operating system as soon as the tasks have finished executing. The operating system is then in charge of executing its own "housekeeping" processes and user applications, as if it would be running on any other device. The soft-PLC and the operating system thus share the same CPU, but the soft-PLC has a higher priority and is executed cyclically. A picture of the device that hosts the soft-PLC of the Mercator TCS is shown in figure 4.8a. It features a powerful industrial computer (quad-core Intel i7 2.4 GHz CPU, 16 GB DDR3L-RAM, CFast 32 GB disk, with built-in UPS) by Beckhoff, installed in the "pumps room" of the building. Pictures 4.8b and 4.8c represent the cabinets close to the telescope (rotating with the azimuth axis), containing the most important electric systems of the TCS. Using our framework, we were able to model these electric systems (consisting of circuit breakers, power supplies, I/O modules, wires, connectors, ...), and then convert the models into human-readable web-pages. An external company then created the electric schematics based on these web-pages, and finally built the actual systems.

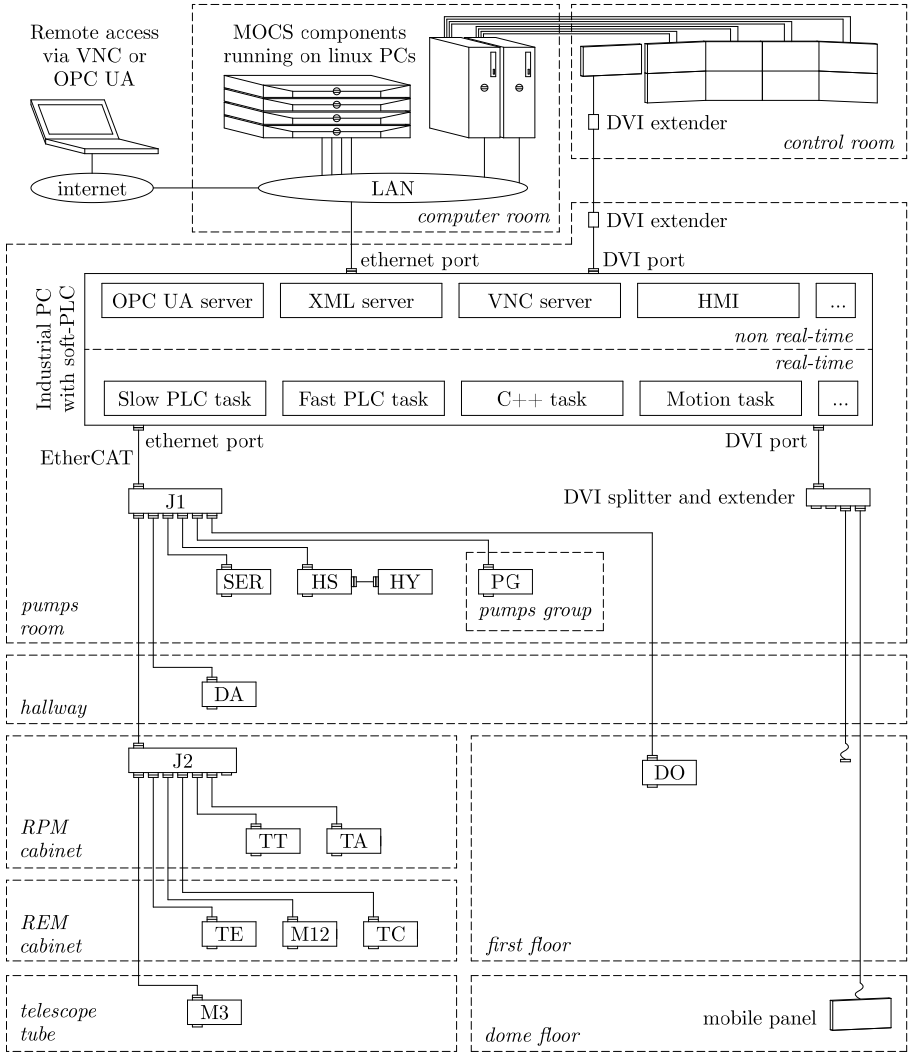
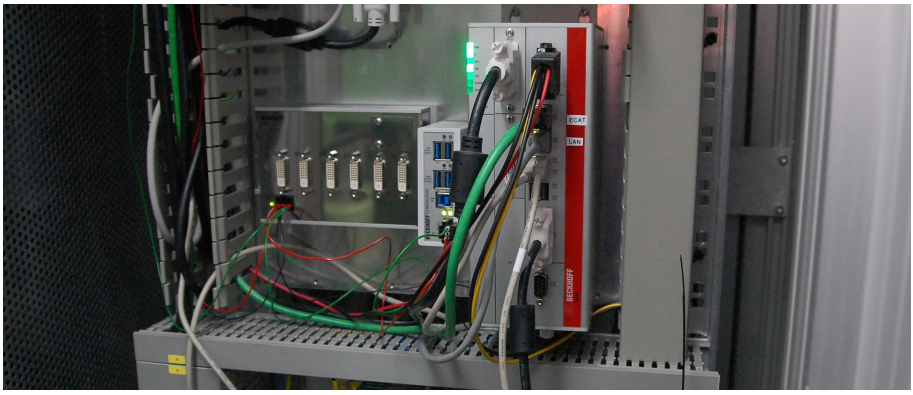


Figure 4.7: TCS architecture.

Several tasks are running sequentially in real-time, distributed over two cores of the CPU. Below we list the most important tasks that may be scheduled every millisecond, in descending priority. They will be referred to in the next section, when discussing the subsystems.

- A TwinCAT NC (Numerical Control) task runs at 500 Hz to control the motion of the telescope axes, M3 and the telescope cover.
- A "slow" PLC task runs the vast majority of the subsystem logic at 100 Hz, except for the logic related to the axes motion control.



(a) The preliminary installed industrial computer that hosts the soft-PLC.

(b) From left to right: the TE, M12 and TC configurations.

(c) The TT configuration (left) and TA configuration (right).

Figure 4.8: Some of the main electric configurations of the TCS.

- A C++ task continuously converts celestial coordinates into horizontal coordinates, and vice versa, taking into account atmospheric refraction and other effects, at 100 Hz.
- A "fast" PLC task runs at 500 Hz, mainly to control the telescope axes and some fast control logic of M3 and M2.

When the above tasks have finished executing, the remaining time of the millisecond is given to the operating system. Below we list the applications that are executed within this millisecond, by the operating system of the Mercator TCS soft-PLC.

- An OPC UA server (TwinCAT OPC UA Server) instance makes all the variables of the real-time tasks (or at least, those marked as readable) accessible over the network via the OPC UA protocol.
- An XML server (TwinCAT XML server) instance provides access of the TCS XML configuration files to the real-time PLC control logic.
- The built-in FTP server of the PLC operating system allows an external back-up service to make daily back-ups of the TCS XML configuration files.
- A HMI (Human Machine Interface, a graphical user interface) instance is running in full-screen on the PLC. It is fully programmed in TwinCAT. The HMI developed for the Mercator TCS is used by both the observer and the technical staff, via a touch panel in the control room and a mobile touch panel in the dome.
- An open-source VNC server instance provides remote access to the desktop of the operating system, and hence, to the HMI instance. Mercator staff uses it to connect to the TCS from outside of the observatory.

At the start or end of the real-time tasks, the soft-PLC communicates with the I/O devices via the EtherCAT protocol. EtherCAT is an efficient real-time protocol: it defines a single master which determines the sequence of bits that can be put on the bus, thereby avoiding the possibility of "collisions" between bits. This is very different from typical multi-access protocols (such as TCP/IP or UDP/IP), which require additional bits for fault detection and recovery. In an EtherCAT network, all interface information of each slave – i.e. its address, the data it can send, and the data it can receive – is known before the communication starts. This enables the master to efficiently address all slaves sequentially, to provide them with data if needed, and to offer them the time to put their data on the bus if needed.

Using the EtherCAT protocol, the soft-PLC receives input data and sends output data from/to the electronics of the subsystems. Each "block" visible in figure 4.7 represents a group of EtherCAT slaves (I/O modules, drives, drive I/O cards) that belong to one or more subsystems. Each group of EtherCAT

Table 4.1: Overview of the TCS electric configurations.

Symbol	*Used by subsystem*	*Summary*
J1		First EtherCAT junction (star point).
SER	Services (4.3.1)	Time service, meteo service, air conditioning service, ...
HS	Hydraulics (4.3.2) and Safety (4.3.3)	Regular I/O to control the hydrostatic bearing of the telescope, safety I/O for the emergency stops and critical hydraulics sensors.
PG	Hydraulics (4.3.2)	Pumps Group (PG) field I/O (regular and safety) to read pressure switches and other sensors of the hydraulics.
HY	Hydraulics (4.3.2)	Drives and contactors of the pumps of the hydrostatic bearing.
DA	Safety (4.3.3)	I/O of the dome access (DA) cabinet, controlling safe access to the dome area.
J2		Second EtherCAT junction (star point).
TT	Telemetry (4.3.4)	I/O connected to temperature sensors, humidity sensors, accelerometers, ...
TA	Telescope axes (4.3.9)	EtherCAT drives and their integrated (safety and regular) I/O, to control the telescope axes.
TC	Telescope cover (4.3.5)	I/O to control the opening and closing of the telescope cover.
M12	M1 (4.3.6) and M2 (4.3.7)	I/O to control the pneumatic primary mirror support, and the custom electronics of the secondary mirror support.
TE	Telescope axes (4.3.9)	Telescope encoders (TE) I/O, to read the interpolated incremental encoders and the absolute encoders of the azimuth and elevation axes.
M3	M3 (4.3.8)	I/O and miniature CANopen drives to control the tertiary mirror.
DO	Dome	I/O and drives to control the dome shutter and rotation.

slaves is contained by an electric configuration (`elec:Configuration` according to our *electricity* ontology), which also contains power supplies, circuit breakers, connectors, etc. of one ore more subsystems. Some of these configurations were already shown in figure 4.8. Table 4.1 provides an overview of the electric configurations of the TCS.

4.3 Subsystems

In this section we will describe the subsystems of the TCS, as designed and implemented using OntoManager, and as installed and commissioned at the observatory during the PhD project. For each subsystem, we will list the most characteristic requirements, and describe the system that we built to satsify these requirements.

4.3.1 Services

The *services* subsystem currently consists of a *time service*, which goal it is to provide an accurate absolute time reference signal to the other subsystems. Its design is mostly driven by two requirements:

R1 *UTC time of time-critical I/O signals is accurate to 1 ms or better.*
R2 *UTC time of the program execution is accurate to 1 ms or better.*

An example of time-critical I/O signals are the encoder readings of the telescope axes. Because multiple encoders are used per axis (e.g. 4 for the azimuth axis), we can instruct the I/O to latch those encoders at a specific time (due to R1), compare this time with the time of the coordinate transformation calculations (due to R2), and therefore send accurate position control corrections to the telescope drives.

To achieve the required accuracy, the TCS must be synchronized with a GPS time reference clock. Instead of relying on a PPS (Pulse Per Second) signal (as it was done by the original TCS), we implemented a more modern "plug-and-play" solution by using IEEE 1588, the Precision Time Protocol (PTP). More specifically, we use a Meinberg M600 rack-mounted time server to provide a IEEE 1588-2008 (PTPv2) master clock signal to a Beckhoff PTPv2 slave I/O module (EL6688, part of the SER electric configuration). The EL6688 module acts as a reference clock for the EtherCAT network. Using the "Distributed Clocks" (DC) feature of EtherCAT, we can synchronize other EtherCAT slaves (such as the encoder modules and the drives of the telescope axes) and the EtherCAT master (the soft-PLC) to this reference clock with an accuracy better than 1 μs [13]. To maintain such a high accuracy, the internal clocks of the EtherCAT master and slaves are periodically synchronized with the internal clock of the EL6688 module, which in turn is periodically synchronized to the GPS clock via the PTPv2 protocol with an accuracy of a few microseconds (e.g. as tested by ESO [48]). The combined accuracy of the DC and IEEE 1588 synchronizations is thus much better than the required 1 ms. Using an RS-232 serial protocol, the TCS additionally reads diagnostic information about the M600 time server, such as the number of satellites visible to the roof-mounted GPS antenna.

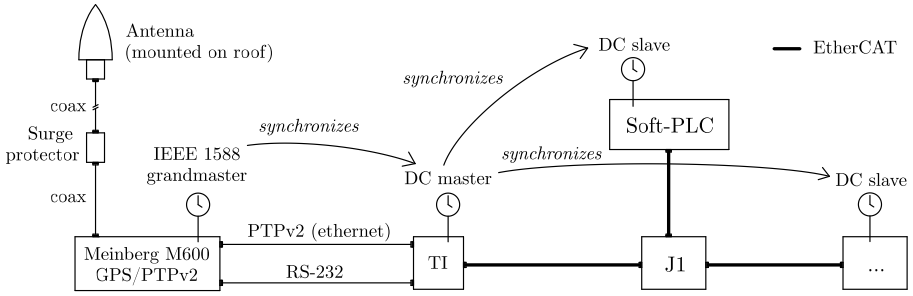


Figure 4.9: Schematic overview of the time reference subsystem.

4.3.2 Hydraulics

The hydraulics subsystem controls the hydrostatic bearing of the azimuth axis of the telescope. In hydrostatic bearings, the solid sliding surfaces are separated by a fluid film, resulting in very smooth moving parts due to the complete absence of sticking friction [93]. In case of the Mercator Telescope, the azimuth axis bearing must support the whole telescope (i.e. the platform, fork, tube, optics, ...) weighing over 8 tonnes, making a hydrostatic bearing an attractive choice. The bearing consists of a steel disk of about 80 cm in diameter, on which the telescope is mounted. The edges of the disc are beveled at an angle of 35°, and are clamped between two fixed rings, with a maximum spacing of 50 μm between the disk and any of the two rings. When oil is injected in the four milled pockets of each ring, a thin layer of oil starts flowing between the two rings and the disk, allowing the telescope to move very smoothly in azimuth direction.

As elaborated in [80], the temperature of the oil flowing through the hydraulic system is not constant: it rises when the system is pressurized, and it varies during the night and during the year due to changing ambient temperature. Variations in oil temperature lead to variations in dynamic viscosity, which lead to pressure variations inside the bearing, and finally to thickness variations of the oil film. To make sure that the oil films inside the bearing (i.e. between the bottom ring and the disk, and between the top ring and the disk) remain of constant thickness, we must therefore add a control system. A chiller (installed inside an annex building) first cools the oil to increase its dynamic viscosity. Any subsequent variation of the oil pressure (due to changing viscosity, due to changing temperature) can be compensated by varying the speed of the pumps of the hydraulic system. For each circuit (i.e. for the top ring circuit and bottom ring circuit) the TCS thus has to control the speed of the pumps based on the temperature of the oil and the theoretic relation between the temperature and the pressure of the oil films. This relation can be approximated by a two-degree polynomial, as derived earlier in my master's thesis [80], and is now implemented in our soft-PLC application. To disturb the tracking of the telescope as little as possible, the pumps speed is not being changed continuously, but with a hysteresis of 5%.

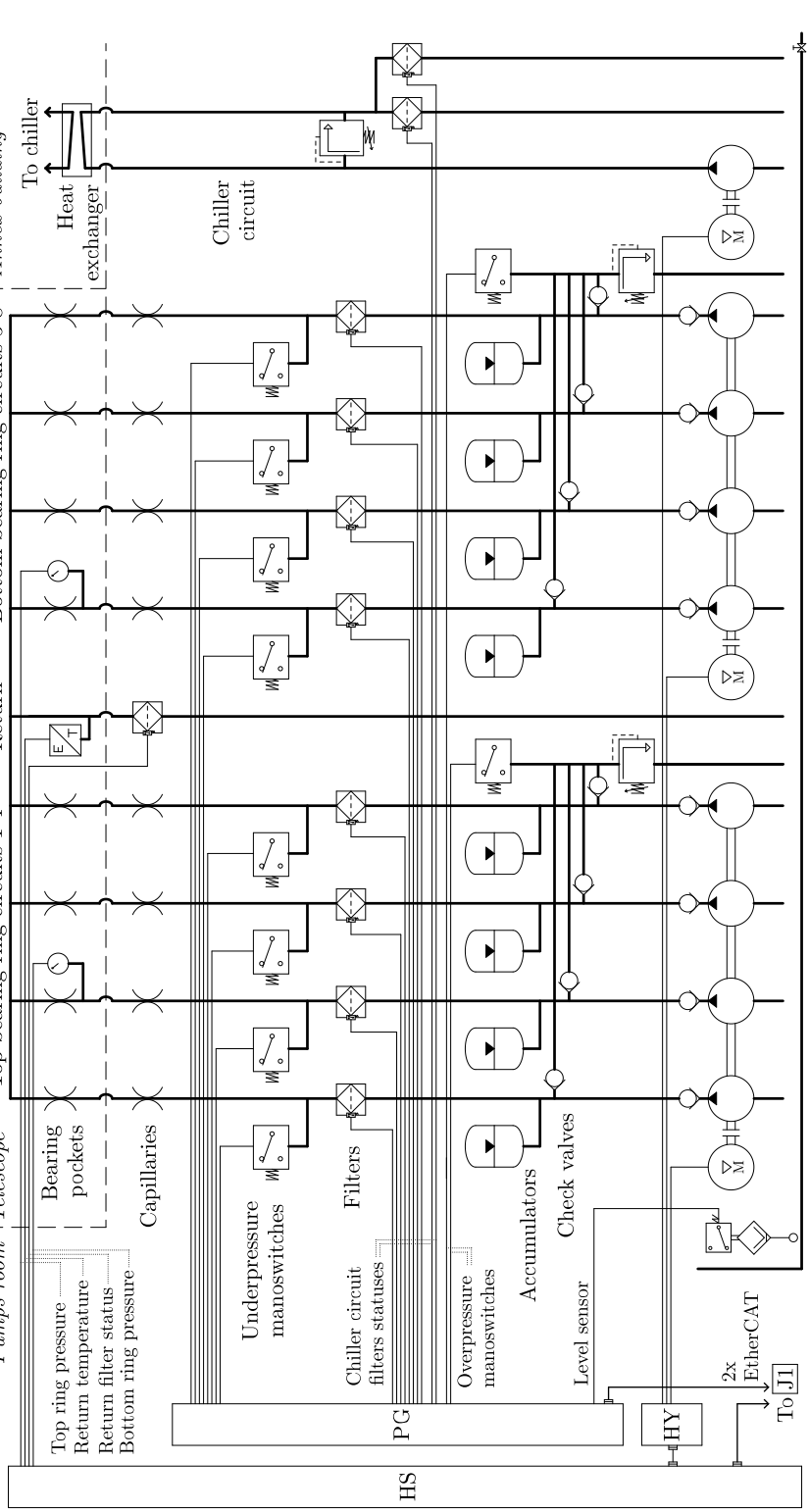
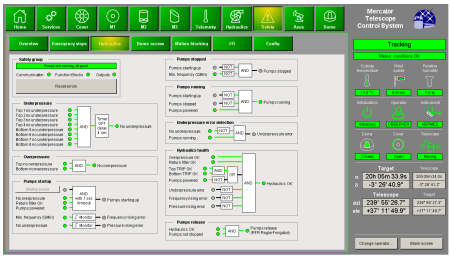


Figure 4.10: Schematic overview of the hydraulics subsystem.

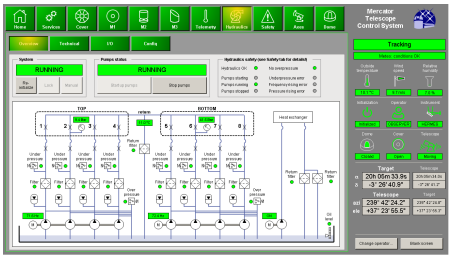
Figure 4.10 represents a schematic overview of the hydraulics subsystem. It consists of three pumps: a circulation pump with on/off control to circulate the oil through the heat exchanger of the chiller, and two pumps with variable speed control to pressurize the oil flowing through the bottom and top ring circuits. At startup, the pressurizing pumps are first run at maximum speed for 10 seconds, to build up the oil pressure. Only if there is no indication of a problem after 10 seconds, then the speed of the pumps is regulated in open-loop according to the measured temperatures and the theoretic relation between temperature and pumps speed. Problems can be underpressure (sensed by 8 manoswitches), overpressure (sensed by 2 other manoswitches), fault signals of the pumps drives, filter throughput problems (sensed by manoswitches in parallel with the filters), and so on. The pumps velocities are controlled by non-safety logic implemented on the PLC, but the system is monitored by the safety logic of the TwinSAFE safety processor (see 4.3.3). The "enable" signals of the pumps drives, and all manoswitches and critical sensors of the hydraulic system, are connected to the safety system via safety I/O. This allows the safety system to intervene if a problem occurs, to avoid damage to the bearing when the telescope azimuth axis would be actuated without sufficient oil pressure. The safety logic responsible for the integrity of the hydraulics system is represented visually by the telescope HMI, see figure 4.11c.

(a) The HS configuration.

(b) The pumps group.



(c) Visualization of the safety logic.



(d) Visualization of the hydraulics system.

Figure 4.11: The hydraulics control system.

4.3.3 Safety

The primary responsibility of the safety system of the telescope is to prevent any movement of the telescope axes (azimuth, elevation and instrument rotation) and the dome when the system has been put in a safe state:

R1 *A 'safe' state exists to prevent all motion of the telescope axes and dome.*

R1 prevents unexpected motion, e.g. while a technical staff member is working at or near the telescope. It has been satisfied by safety certified components only (which was not the case for the original TCS). The 'safe' state can be entered by pressing one of the emergency stop buttons which are installed at various places inside the observatory. Always the nearest emergency stop button should be pressed, to avoid other persons from releasing the button. When not in the 'safe' state, slow movements of the telescope, instruments and the dome should always be expected. This does not lead to hazardous situations, however, because:

R2 *Normal operations shall not involve approaching dangerously moving objects.*

'Normal operations' are considered to consist of the daily tasks of the observer while the system is not experiencing technical problems. Nearly all of these tasks are handled within the safe environment of the control room. During normal operations, the observer only needs to enter the dome area to open or close the dome sideports (windows) for increasing the airflow inside the dome, a task which does not require approaching the telescope or dome roof. In case of abnormal operations (e.g. if the telescope cover has a problem and needs to be opened by hand), then the 'safe' state should be entered first by pressing the nearest emergency stop button. These (and other) safety guidelines are available on-line at the Mercator website and are instructed to observers by the technical staff. It is the responsibility of the person inside the dome area to respect them (e.g. turn on the lights inside the dome and do not approach or touch moving parts unless an emergency button is pressed) so that hazardous situations cannot occur. Neglecting to follow the safety guidelines does not necessarily lead to hazardous situations however since:

- Large moving objects such the telescope axes, instrument derotators and dome roof do not move or accelerate fast (see table 4.2).
- Whenever possible, dangerous parts of the system (such as spur gears or electric circuitry) require tools to be accessed. In seldom cases (such as for the large rack gear around the dome circumference) this was not possible, however.
- Since June 2016, a 'dome access control' system has been installed, which automatically turns on the lights inside the dome and puts the system in the 'safe' state while a person is inside the dome area (see further).

Object	Max. acceleration	Max. velocity
Telescope azimuth axis	$1.5°/s^2$	$3°/s$
Telescope elevation axis	$1.5°/s^2$	$3°/s$
Telescope instrument derotator axes	$3°/s^2$	$6°/s$
Dome rotation	$1.5°/s^2$	$3°/s$

Table 4.2: Maximum accelerations and velocities of large objects.

If an emergency stop button is pressed while parts are moving, then the moving parts will perform a 'Safe Stop 1' (SS1). In this case the non-safety program will initiate a controlled stop along a 'quick stop' ramp, to stop the axis as fast as possible while respecting the mechanical constraints (e.g. slipping torque of the gear train) and electrical constraints (e.g. peak currents). The SS1 is terminated when the safety system enters the STO (Safe Torque Off) state, at a fixed delay after the emergency stop button has been pressed, regardless of whether the controlled quick stop has been successful or not. The optimal quick stop ramps and the delays of the STO state have been determined experimentally. They are controlled by safety logic, running on the Beckhoff EL6900 EtherCAT module and the AX5805 safety drive cards. The EL6900 is an EtherCAT slave, but it has a CPU to run safety logic, and it uses the EtherCAT network to communicate to the other safety devices (e.g. safety I/O modules and drive cards) via the TwinSAFE protocol. The safety logic is visualized by the TwinCAT HMI (see figure 4.12).

Since there is no requirement for observers to access the dome area while observations are ongoing, a new measure has been taken in 2016 to further increase the safety of the observers. The 'Dome Access' (DA) control system detects when a person opens the doors to the dome area at the ground floor level, and consequently turns on safety lights inside the dome area and stops the telescope and dome gracefully before forcing the system into the 'safe' state. As a result, it has become impossible for an observer to enter the dome area while observations are ongoing. The safety lights can only be turned off and the 'safe' state can only be left when the doors are closed and an acknowledge button outside the dome is pressed. The access control system is a complementary measure though, it is not sufficient to guarantee safety on its own because:

- It can be (permanently or temporarily) disabled by technical staff, locally or remotely, using only a password.
- It has been partially built with non safety-certified (not fail-safe) components.
- It can be bypassed (intentionally or unintentionally) by two persons: a person outside the dome area can press the acknowledge button while another person remains inside.

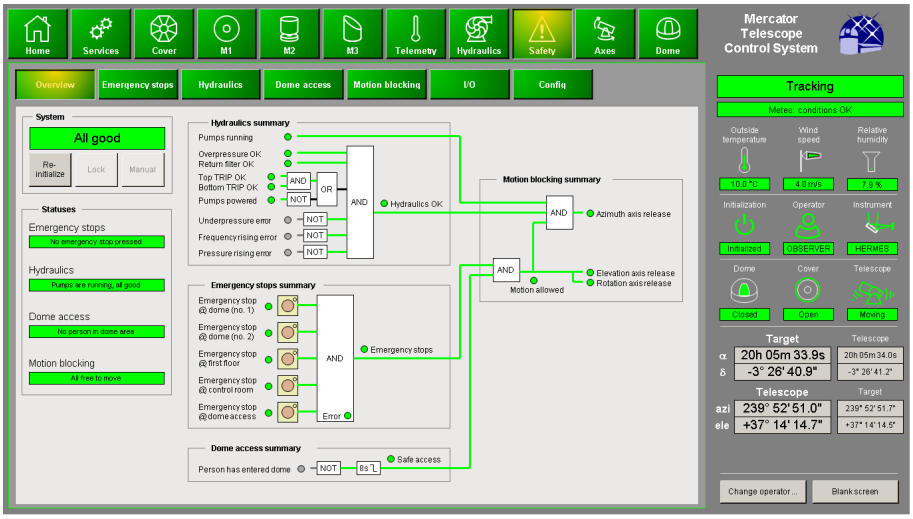


Figure 4.12: Visualization of the safety control system.

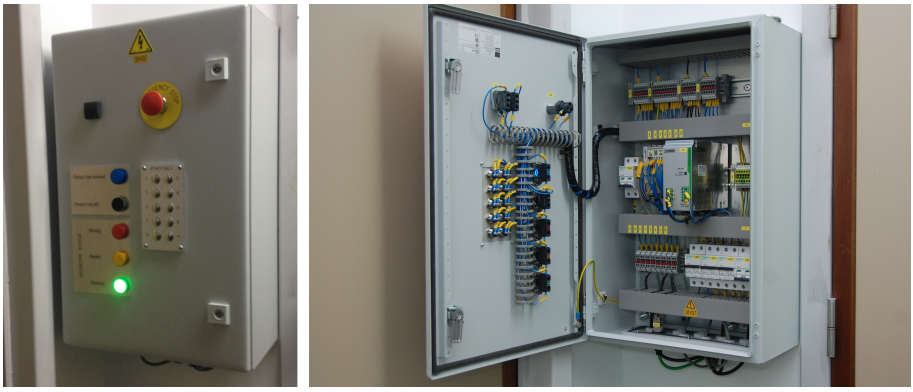


Figure 4.13: The dome access cabinet.

4.3.4 Telemetry

The responsibility of the telemetry subsystem can informally be defined as the data acquisition of sensors that are not directly electrically connected to other subsystems. The telemetry subsystem has dedicated I/O electronics to gather the data of several sensors mounted on various locations at the telescope and the building. Its software model defines a `Telemetry` object that acts as a shared resource for other subsystems. For instance, the M2 subsystem uses some telemetry sensor values to predict the position setpoint of the secondary mirror, along the optical axis of the telescope, for achieving the optimal focus for a given instrument. The telemetry subsystem currently measures 13 temperatures, 2 relative humidities, and the elevation angle of the telescope via accelerometers (independently of the telescope axes subsystem).

4.3.5 Telescope cover

The primary intent of the telescope cover is to protect the mirrors of the telescope from environmental hazards inside the closed dome. It was originally conceived as a lightweight canopy made of fabric, put on or taken off the telescope tube manually by the observer, but it has been replaced in 2014 by an electrically actuated solution. Its main requirements are listed below:

R1 *A 'closed' state prevents dust from entering the tube.*
R2 *An 'open' state does not interfere with observations.*
R3 *Switching between 'closed' and 'open' states takes < 2 min.*
R4 *'Open' and 'closed' states are manually reachable without electrical power.*
R5 *The cover is resilient to 'high' wind load while in 'open' or 'closed' states.*
R6 *Heat dissipation < 30 W while in 'closed' or 'open' state.*
R7 *No noisy current consumption while in 'closed' or 'open' state.*

The chosen design consists of two sets of four aluminum petals, which slightly overlap when pressed against the M2 enclosure to seal the tube (R1). The cover is then closed, as shown in figure 4.15a. To open the cover, the petals can rotate for about 250° until they are aligned with the tube (R2, see figure 4.15b). Each petal is actuated by an electric motor via a gear transmission and a mechanical clutch, and receives position feedback from an absolute encoder. The gear reduction was selected such that the panels can open and close in reasonable time (R3) while limiting the required torque and the resulting size of the motors. Because the gear train is non-reversible and the aluminum petals can slightly be deformed elastically, the petals will maintain their position when they are pushed against the tube or the M2 enclosure and the electric actuation is removed. When a higher torque is applied, the mechanical clutch will slip so that the petals may be moved manually even without the electric actuation (R4). To guarantee resilience to high wind loads, electromagnets were installed on the telescope tube to pull the petals even stronger against the tube while they are fully open (R5). The magnets consume only little DC power which cannot interfere with the detector readout of a nearby instrument (R6, R7).

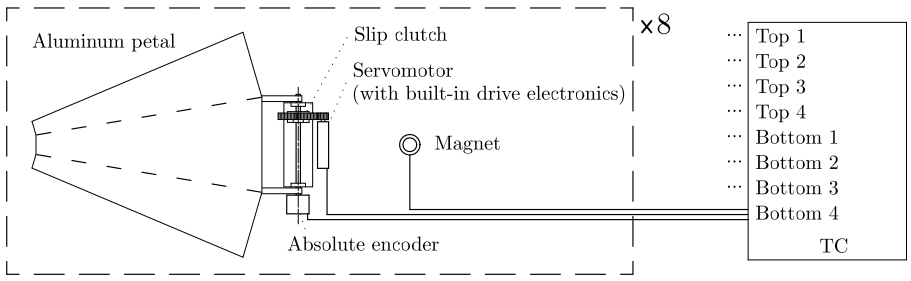
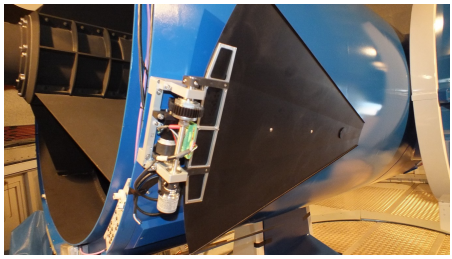


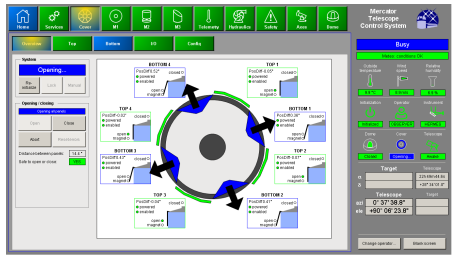
Figure 4.14: Schematic overview of the telescope cover.

(a) Closed state.

(b) Open state.



(c) A single panel.



(d) Visualization of the cover.

Figure 4.15: The telescope cover.

4.3.6 M1: Primary mirror

The primary mirror (M1) of the telescope collects the parallel light rays from the sky and reflects them at an angle (due to its hyperbolic shape) to the secondary mirror. The Mercator Telescope primary mirror is a solid Zerodur disk with a thickness at the edges of 150 mm, with a concave hyperbolic reflective side, and a central hole of 330 mm to pass the light from M2 to the Cassegrain focal station. The shape of the disk, and thus the shape of the hyperbolic reflective surface, is distorted by gravity depending on the elevation of the telescope tube. The primary task of the M1 control system is to maintain the shape of the reflective surface by counter-acting this gravitational distortion. Some of its most characteristic requirements are listed below:

R1 *Axial pressure is applied by optimally distributed pads at the M1 backside.*

R2 *Radial pressure is applied by optimally distributed pads at the M1 circumference.*

R3 *Technical staff is able to enter manual pressure setpoints.*

Since the distribution and the number of pressure pads has been fixed by the original design of the telescope, the control laws of the original TCS were implemented without modification on the new PLC-based TCS. In “auto” mode, both the radial and the axial pressures are controlled in open-loop: the pressure setpoints are calculated based on the tilt of the mirror (measured by

accelerometers of the telemetry subsystem). They can be tuned in software, via the M1 configuration. Technical staff may also enter a "manual" mode (only when they are logged by password), in which fixed setpoints can be sent to the valves (R3). As studied in [7], the current mirror support is not optimal however: gravitational distortion of a tilted concave mirror cannot be fully counter-acted by the current configuration of axially and radially distributed pressure pads. Improvements in the future may include changing the position of the pads, or adding pads which "pull" the mirror from the top (instead of only "pushing" the mirror from the back and the bottom), or applying different pressures for different pads. See figures 4.16 and 4.17.

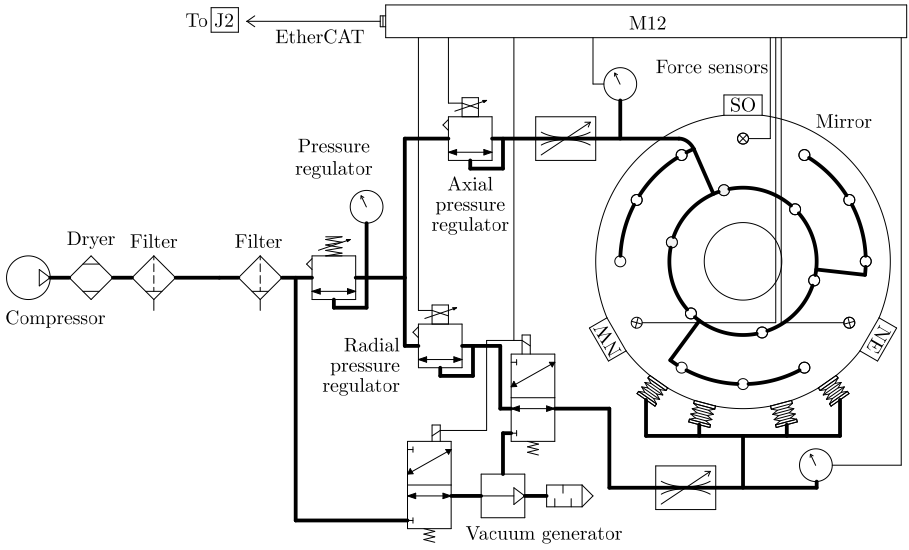
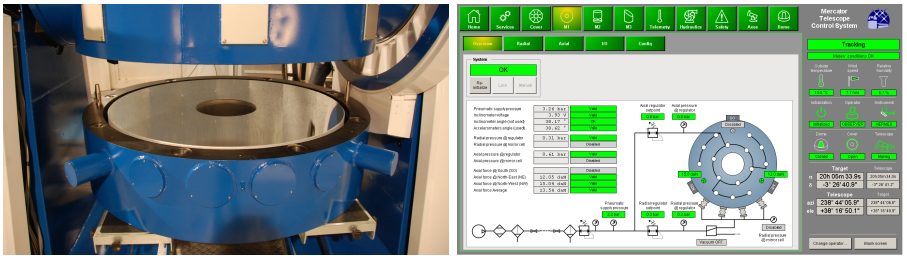


Figure 4.16: Schematic overview of the M1 subsystem.



(a) The primary mirror. (b) Visualization of M1.

Figure 4.17: The primary mirror of the telescope.

4.3.7 M2: Secondary mirror

The secondary mirror (M2) of the telescope is a convex hyperbolic mirror of 307 mm in diameter. It reflects the light coming from M1 to one of the focal stations (either directly trough a hole of M1, or indirectly via M3 through holes of the tube). The mirror is supported by a compact assembly of three translation mechanisms (X, Y, and Z) and two rotation mechanisms (tilt-X and tilt-Y), allowing the optics of the telescope to be optimally aligned. Only the Z axis has to be repositioned several times during the night, to re-focus the light on the instrument detectors, due to thermal expansion of the telescope tube (as a consequence of changing ambient temperature) or due to an instrument change request. A schematic overview of the M2 control system is shown in figure 4.18.

The mechanical assembly and the field electronics (i.e. the miniature drives, multiplexer, power supply, etc. mounted inside the M2 enclosure inside the tube) are shown in figure 4.19a. They have been built by the Geneva Observatory, and are extensively described in [95]. Each axis is actuated by a miniature electric servomotor equipped with Hall sensors, and its position is measured by an external potentiometer (for X, Y, Tilt-X and Tilt-Y) or absolute encoder (for Z). All motors are commanded by a miniature drive, which uses the Hall sensor pulses as motor feedback. Motion is controlled by two digital inputs of the drive: a "brake" input to enable or disable the motion, and a "direction" input to choose the direction of the motion. Only the Z axis uses a third input, to switch between high speed or low speed (to allow faster positioning, since the Z axis can travel 40 times further than than the other axes). A multiplexer reduces the size of the electric interface between the M2 field electronics and the commanding electronics installed inside a cabinet below the telescope platform.

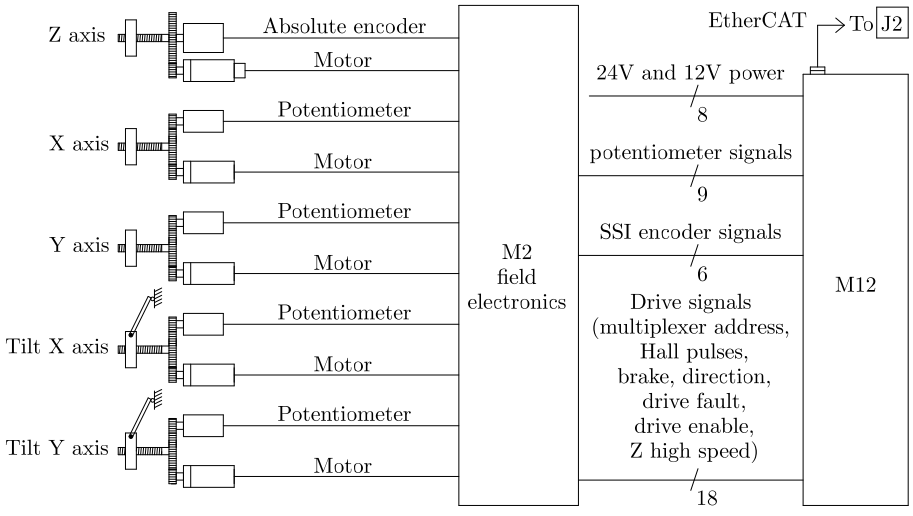


Figure 4.18: Schematic overview of the M2 support.

It allows the common digital signals of the axes (i.e. the Hall sensor signals, the brake signal, and the direction signal) to be multiplexed and transmitted over the same wires, at the cost of adding three signals (A, B, C) to specify the address of the currently selected axis. The field electronics of M2 were not altered when replacing the transputer system – except for the original encoder of the Z axis, which was replaced by a SSI (Synchronous Serial Interface) encoder because the latter is supported out-of-the-box by the Beckhoff I/O system. Only the transputer system has thus been replaced by our soft-PLC and the M12 electric configuration. The main requirements of the original system were maintained by the new TCS:

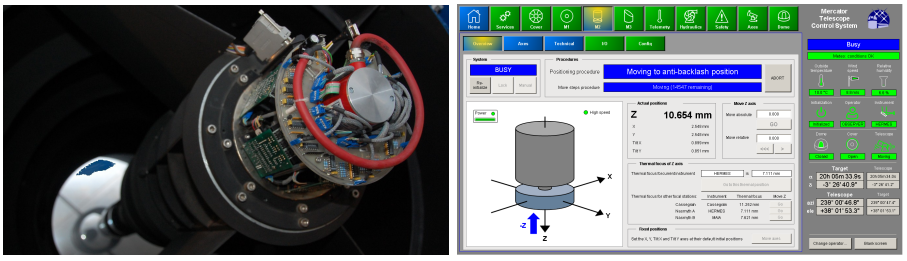
R1 *Final motion of an axis always has the same sense.*

R2 *Final motion of an axis has compensated backlash.*

R3 *Final motion of an axis is opposite to the gravitational vector.*

R4 *Axes are positioned at an accuracy of at least their external encoder resolution.*

The control algorithms of the new TCS differ significantly from the original control algorithms, as they store whether or not the backlash of a given axis was already compensated, between two positioning commands. In many cases, this allows for faster positioning, since no unnecessary movements need to be performed to satisfy R1-3. With "final motion" we mean the last part of a trajectory, when the axis decelerates to a standstill without changing the sense of direction. To satisfy R4, the TCS counts the number of Hall pulses during positioning, since the resolution of the Hall sensor feedback is higher than the resolution of the absolute encoder or potentiometer feedback. The brake signal is set at a very short time before the target position is reached, to allow the system to decelerate during the final milliseconds of the positioning. The "fast" 500 Hz task of the PLC takes care of this, while the "slow" 100 Hz task executes all other M2 logic.



(a) Field electronics of M2. (b) Visualization of M2.

Figure 4.19: The secondary mirror of the telescope.

4.3.8 M3: Tertiary mirror

The tertiary mirror (M3) of the telescope either reflects the light coming from M2 to one of the Nasmyth focal stations, or it sits next to the beam coming from M2 to let the light pass to the Cassegrain focal station. Before installing the MAIA imager in 2012, M3 only had one degree of freedom, to switch between the HERMES spectrograph (at Nasmyth A) and the decommissioned MEROPE imager (at Cassegrain). To let the beam be reflected to the other Nasmyth focal stations (including Nasmyth B, which hosts MAIA), a new support for M3 was developed in collaboration with the Geneva Observatory. This new support consists of a translation stage (to switch between the Cassegrain focal station and the Nasmyth focal stations) and a rotation stage (to rotate the mirror towards one of the four Nasmyth focal stations of the telescope). A schematic overview is shown in figure 4.20.

The M3 support consists of two stages: a translation stage to flip the mirror between Cassegrain and Nasmyth positions, and a rotation stage to rotate the mirror to a particular Nasmyth position. The mirror is mounted on a angled hollow cylinder, which passes the light through when the mirror is parallel to the light beam coming from M2. By turning the lead screw of the translation

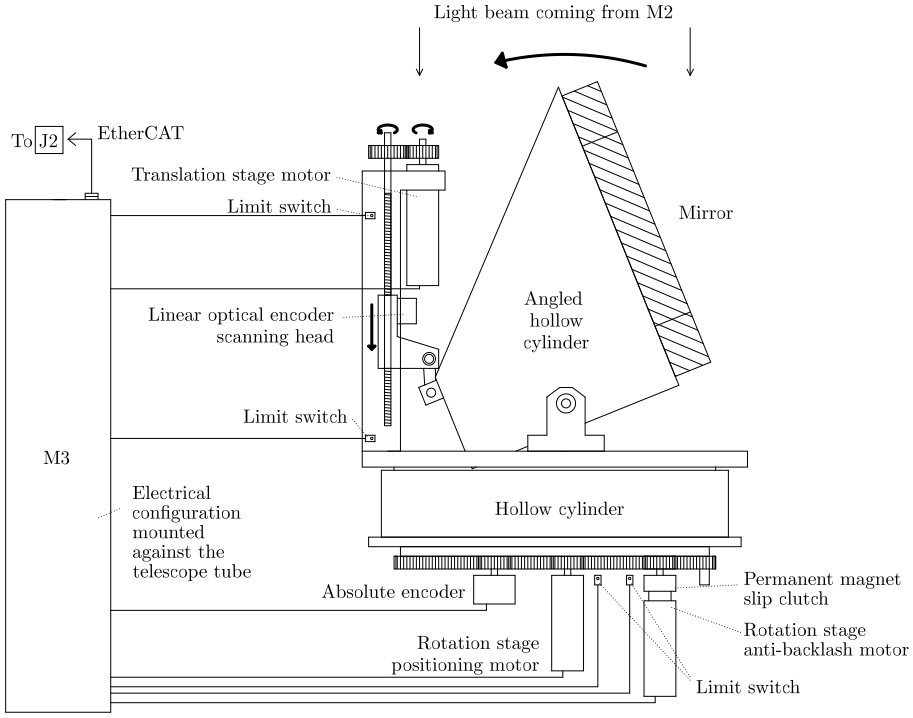


Figure 4.20: Schematic overview of the M3 support while busy switching from Cassegrain to Nasmyth (thick arrows indicate motion).

stage, a nut pulls the angled hollow cylinder until it is tilted at an angle of 45°, to reflect the light beam by 90° out of the telescope tube to one of the Nasmyth focal stations. A particular Nasmyth position is selected by positioning the rotation stage, which is also mounted on a hollow cylinder, to let the light beam pass to Cassegrain if needed. The main requirements of the M3 control system, as derived from the design document[6], are listed below:

R1 *The translation stage can be positioned at a repeatability better than 5.6 μm.*

R2 *The rotation stage can be positioned at a repeatability better than 23 arcsec.*

R3 *Backlash is compensated for the final motion of a stage.*

R4 *No power is needed to maintain a position.*

A precise linear Heidenhain encoder (resolution 1 μm) and rotary Kübler encoder (17 bits), in combination with large gear reductions and precise servomotors, allow the mechanisms to achieve the required repeatability (R1-2). Friction in the lead screw mechanism and the motor gear reductions satisfy R4. Backlash between gears is eliminated by using spring loaded scissor gears. The remaining backlash of the translation stage at the nut is below the required repeatibility, while the remaining backlash of the rotation stage is eliminated by generating a counter-torque via a second motor. To maintain a counter-torque when the motor is not powered, a magnetic clutch can be preloaded before turning off power. The PLC software implements all positioning logic, homing procedures, and a calibration procedure to find the position of the anti-backlash motor when the magnetic clutch is maximally preloaded (see [91]). The drives of the M3 motors are connected via CAN-bus (with CiA 402 drive profile) and a bridge terminal to the EtherCAT network, allowing for easy integration in the Beckhoff TwinCAT 3 software.

(a) The tertiary mirror. (b) The M3 electric configuration.

Figure 4.21: Pictures of the M3 control system.

[6]Document OBSGE-T5-TO-RO1, version 1, Feb 2010.

4.3.9 Telescope axes

The *telescope axes* subsystem is responsible for the motion of the azimuth axis, the altitude axis, and the instrument derotators of the Cassegrain and Nasmyth B focal stations. It allows the telescope to be positioned in either:

- **encoder coordinates**: azimuth (*azi*) and elevation (*ele*) angles, according to the encoders;
- **equatorial coordinates**: right ascension (α) and declination (δ) angles, on the sky.

To convert from equatorial coordinates to encoder coordinates (or vice versa), an "intermediary" conversion to the **true horizontal coordinates** is needed. The latter are defined as the angle between the optical axis of the telescope and the North, and the complement of the angle between optical axis of the telescope and zenith, respectively. True horizontal coordinates differ from encoder coordinates since they take into account the flexure of the telescope tube, the offsets between the encoders and North and zenith, the imperfect perpendicularity of the axes, and so on. Conversion from encoder coordinates to true horizontal coordinates (and vice versa) requires the application of a mathematical model of these error sources. At the Mercator Telescope, we use the *TPOINT*[7] model by P. T. Wallace to correct for the largest error sources.

Conversion from true horizontal coordinates to equatorial coordinates (and vice versa) is also affected by several sources of error. Many of them are predictable and can be modeled, such as:

- precession and nutation: the long-term and short-term changes of the Earth's rotation axis with respect to the ecliptic (the apparent path of the Sun on the sky) due to gravitational interactions between the Earth and the solar system bodies;
- atmospheric refraction: error in elevation due to refraction of the light of celestial targets by the atmosphere (which depends on the ambient temperature, pressure, relative humidity, observed wavelengths, ...);
- annual aberration: apparent displacement of celestial targets due to the Earth's orbiting velocity around the Sun and the finite speed of light;
- parallax and proper motion: apparent displacement of a nearby star due to the Earth's orbit around the Sun (parallax) and due to the velocity of the nearby star with respect to the Sun (proper motion).

At the Mercator Telescope, we correct for the above (and other) sources of error via the SLALIB positional astronomy library by P. T. Wallace [121]. Another source of error is not easy to predict: the deviation called *Delta T* between

[7] TPOINT software homepage: `http://www.tpointsw.uk`.

the Earth's rotation (UT1) and the Coordinated Universal Time (UTC). Delta T changes slightly with time due to variations in the rotation of the Earth. Whenever it reaches +/– 1 second, a leap second is added to (or subtracted from) UTC, to keep UTC and UT1 aligned to within +/– 1 second. The *time service* subsystem of the TCS provides the time to the *telescope axes* subsystem as UTC (which is the sum of the International Atomic Time (TAI) and the actual number of leap seconds, both of which are provided by our GPS clock). Delta T can be entered in the configuration of the TCS if very accurate equatorial coordinate conversions are needed. It can be retrieved from the International Earth Rotation and Reference Systems Service (IERS), online at `http://maia.usno.navy.mil`.

Knowing the largest sources of error allows us to understand the measures which need to be taken to satisfy the most important performance requirements of the *telescope axes* subsystem:

R1 *Azimuth and elevation positioning is repeatable to +/- 0.2 arcseconds.*

R2 *Pointing to equatorial coordinates is always better than +/- 0.5 arcminutes.*

R3 *Tracking errors are less than 0.3 arcseconds RMS during 30 seconds intervals.*

R1 is only about repeatability: it determines the precision of the encoders and the maximum position error of the azimuth and elevation axes according to these encoders. Positioning can be absolute (e.g. when parking the telescope at specific (azi, ele) coordinates) or relative (e.g. when sending a guiding correction as $(\Delta azi, \Delta ele)$ angles to the TCS).

R2 is about the "absolute" accuracy of the positioning of the optical axis of the telescope. It takes into account the conversion errors between equatorial and true horizontal coordinates (implemented by SLALIB) and the conversion errors between true horizontal and encoder coordinates (implemented by TPOINT).

R3 is about the ability of the axes to accurately track an object on the sky, which appears to be moving due to the Earth's rotation. When such a target appears to move trough zenith, its apparent azimuth velocity becomes infinitely fast – or at least very fast if it moves close to zenith. To respect the maximum velocity and acceleration of the azimuth axis, we therefore limit the tracking range of the telescope to an elevation angle below 89°. The azimuth velocity when tracking objects which are close to 89° elevation is thus acceptable, but it is still high, leading to a deterioration of the tracking performance. R3 is satisfied as long as this deterioration doesn't exceed 0.3 arcseconds within a 30 seconds interval. A gradual increase of the deterioration on a longer time-scale is considered not harmful, since the guiding control loop (running typically at a cycle time of 5 seconds) can correct for this increase.

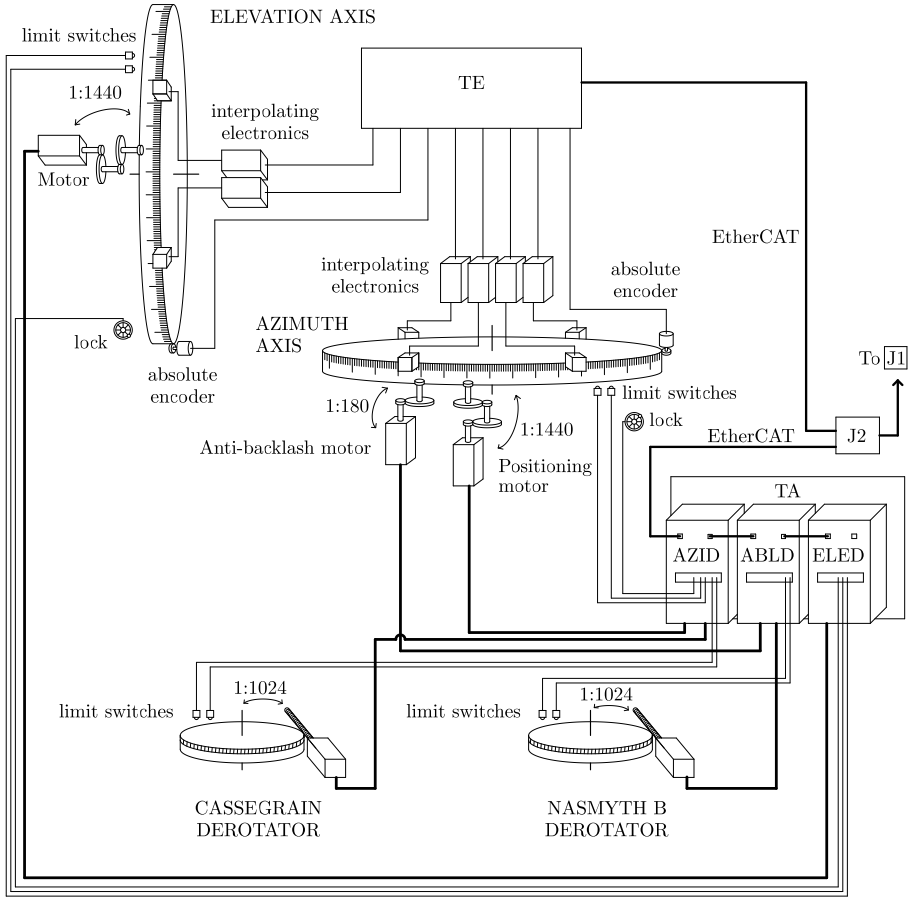


Figure 4.22: Schematic overview of the telescope axes subsystem.

Figure 4.22 shows the main components of the telescope axes subsystem. Two electric configurations are directly connected to the field devices: TE (Telescope Encoders) holds the I/O to read out the encoders, while TA (Telescope Axes) holds three dual-axis drives. These drives are equipped with TwinSAFE safety cards, to allow all safety aspects to be controlled by the safety subsystem using the existing EtherCAT network. The system consists of four axes:

- The **azimuth** axis is driven by a “positioning” motor and an “anti-backlash” motor. The latter provides a variable counter-torque to eliminate backlash in the gear train of the former. Both motors are equipped with high-resolution encoders and brakes, allowing the positioning motor to be controlled at a very fast rate in position control mode by the AZID drive (see further). An external incremental encoder is used to continuously "correct" the position of the motor encoders. It consists of four optical scanning heads and a common linear encoder ribbon wrapped around

the telescope axis. The scanning head signals are interpolated 100 times to increase resolution, thereby converting 11 μA sinuses to square TTL-level pulses. These pulses are read by four I/O modules, which are simultaneously latched via the SYNC pulse of the Distributed Clocks functionality of EtherCAT. The latched encoder values are finally averaged in software, to eliminate eccentricity and discontinuities of the wrapped linear encoder ribbon. Two limit switches and a "lock" switch (enabled when the axis is physically blocked by a screw mechanism) are directly connected to the digital inputs of the drive. An external absolute encoder is only used during the homing procedure, to bring the axis close to a "known" homing mark of the linear encoder ribbon.

- The **elevation** axis is very similar to the azimuth axis, but it only has two encoder scanning heads, and backlash is eliminated by the torque which results from deliberately unbalancing the telescope tube.
- The two **derotator** axes have much lower repeatability and accuracy requirements since they only have to compensate field rotation (and a field covers 360° in rotation, whereas it only covers a few arcminutes in azimuth and elevation). They are driven by a worm-gear mechanism. The motor of this mechanism is equipped with a absolute encoder, sufficiently accurate to position the axis.

The original motors, drives, absolute encoders, and I/O modules were replaced when installing the new telescope axes subsystem (see figure 4.23). This allowed us to achieve a higher performance because, as will be detailed below, the new electronics allow us to close the position control loop 10 times faster than before, because "ideal" positioning trajectories (S-curves) are calculated instead of relying on slow PID algorithms to introduce small guiding offsets, and because the 17-bit encoders of the newly installed motors are much more repeatable than the original resolvers and potentiometers (which generally have a repeatability of a few percent). Since the electronics are never switched off, the positions of the incremental encoders are never "lost", and therefore no time-consuming homing

(a) The dual-channel motor drives, equipped with safety cards.

(b) The azimuth positioning and anti-backlash motors.

Figure 4.23: The drives and two motors of the axes control system.

procedures need to be performed when the telescope is put in its operational state at the beginning of every night. This has reduced the startup time of the telescope to less than 1 minute (compared to over 10 minutes of the original TCS).

The system also looks “cleaner” since much less cables and wires are needed. For instance, safety is controlled by the software in the safety cards of the drives (there is no hard-wiring), drives can communicate with each other via the fast EtherCAT network instead of via dedicated signal wires, and motors are connected via Beckhoff’s so-called “one-cable technology”. Also, the original wired “remote control” box was replaced by a mobile touch panel, to allow access to all functionality of the TCS from anywhere in the dome.

While the full control logic is too complex to be described briefly in this thesis, we explain a small part of this logic using figure 4.24. The figure shows how the TCS receives a pointing command, via OPC UA or via the HMI. The TCS first sends the new equatorial coordinates to the C++ task, which converts them to the true horizontal coordinates. Focusing only on the azimuth axis, the “true” azimuth position, velocity and acceleration are sent to the “fast” PLC task. Using the velocity and acceleration, the true azimuth setpoint position is estimated, and converted to the encoder reference frame via the TPOINT model. A new pointing command consists of a pointing phase (i.e. move as

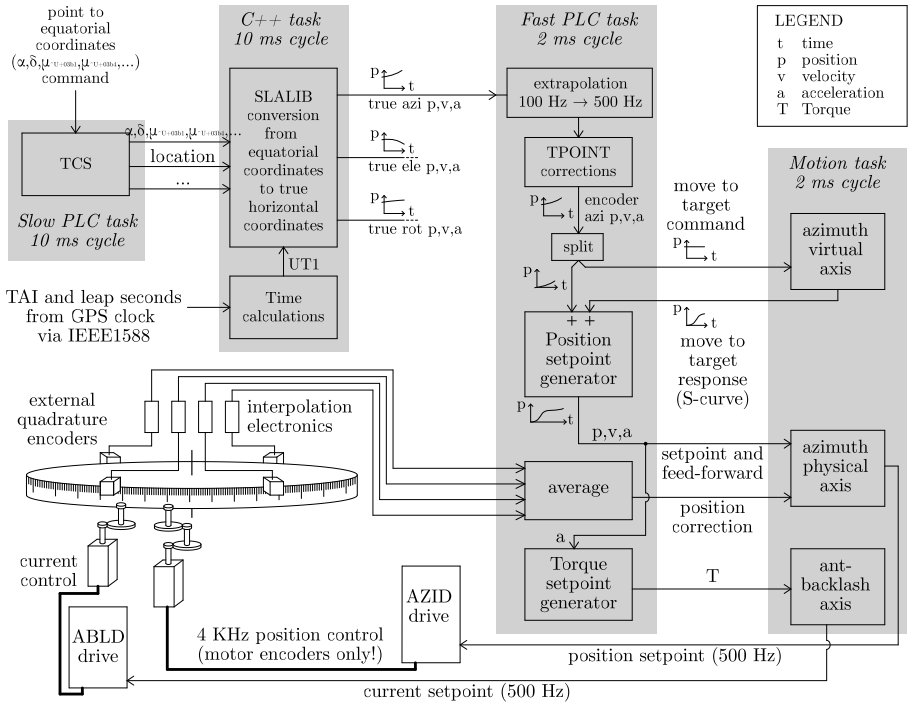


Figure 4.24: Schematic overview of the telescope axes subsystem.

Figure 4.25: Visualization of the axes control system.

fast as possible to the target) and a tracking phase (i.e. follow the target). In original TCS (as in many TCS'es), these phases were executed in sequence, leading to a positioning error right after the pointing phase due to the apparent motion of the star during the pointing phase. Correcting this positioning error is slow, because it typically depends on the PID (Proportional-Integral-Derivative) algorithm of the position controller. In the new TCS, we therefore "split" the relative apparent motion of the star, and the initial pointing motion of the telescope. The TCS executes the pointing and tracking phases in parallel: it starts to track the apparent motion of the star, relative to initial position of the axis, as soon as the pointing command is received. Because the telescope is now tracking the star at constant distance, we can send this constant distance to a "virtual axis" (not connected to a physical drive) of TwinCAT, which converts the step input into a smooth S-curve output signal. By adding this output (i.e. position, velocity and acceleration) to the relative motion (i.e. position, velocity and acceleration) of the target, we can now send the optimal trajectory to the physical axis, and finally the drive. Also any subsequent positioning commands (such as guiding offsets) can be given to the virtual axis, leading to fast S-curve responses instead of relying on slow PID actions. The "positioning" motor of the azimuth axis is running in "position control mode" at 4 kHz – much faster than the original drive which was configured in "speed control mode" at 50 Hz. The new TCS still has to correct these "motor encoder positions" however at a lower frequency (500 Hz), by averaging the four interpolated external encoder scanning head signals. As can be seen from the figure, also the torque command of the anti-backlash drive is updated at 500 Hz. A torque setpoint generator calculates the optimal torque of the anti-backlash motor, based on the acceleration of the positioning motor. Optimal means that the counter-torque applied to the positioning motor always has the same sense – also during acceleration or deceleration of the axis.

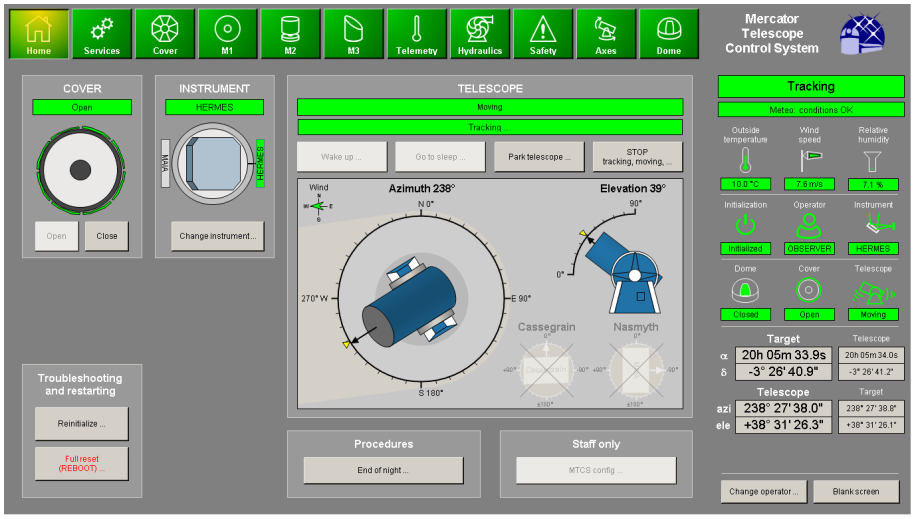


Figure 4.26: Overview screen of the TCS visualization.

The control logic of the telescope axis is much more complex than what is shown in figure 4.24, however. Not only basic functionality (such as homing procedures, relative positioning, horizontal coordinate positioning, safety procedures, conversion from the actual encoder coordinates to equatorial coordinates, ...) but also more advanced functionality (such as relative positioning in $(\Delta\alpha, \Delta\delta)$ angles via optimal S-curves, tracking solar system objects, calculation of the optimal path of an axis with respect to the cable wrapping, ...) has been implemented. Describing this functionality is out of the scope of this thesis, but it is well documented in software – and, of course, in the web pages generated by OntoManager.

4.4 Summary

In this chapter, we described the new telescope control system (TCS) of the Mercator Telescope, the user interface of which is shown in figure 4.26. By replacing the original software and hardware (i.e. transputers and Solaris machines) with a new system that consists of a soft-PLC and an EtherCAT I/O system, we were able to apply the knowledge-driven methodology proposed by this thesis. The focus of this chapter has been on the application itself: it provides a description of the Mercator TCS architecture and a functional description of the subsystems that were implemented using OntoManager. With the experience of applying the framework in mind, we can now better evaluate the framework in the next chapter.

Chapter 5

Evaluation

HAVING experienced the application of our framework to a real-world control system in the previous chapter, we can now refer back to the original framework requirements that were outlined in chapter 2. By going through these requirements, we can evaluate the framework qualitatively (and quantitatively if possible) by identifying the most important functional and/or performance issues related to each requirement. Any reference such as R1, R1.1, R1.2, etc. refers to the requirements listed in 2.3.

5.1 Ontologies

The first requirement (R1) is about ontologies, which *"shall formally specify the semantics of the domains of interest for the development of telescope control systems."* Looking at the extensive list of ontologies in 3.3, clearly a wide range of domains of interest are covered – although a *telescope* ontology or *astronomy* ontology has not been created. The reason is that we have focused on the "lower-level" parts of the system, mostly to facilitate the design of the electric cabinets and the control software. A *telescope* or *astronomy* ontology would be more useful at a higher level, e.g. to describe the information of a celestial target (which must be exchanged between a scheduler application and the TCS), to describe the "observing modes" of the observing system (which define the state of the mirrors and the instrument derotator axes), to describe states of the system that involve celestial objects (such as the state in which the telescope is pointing dangerously close to the Moon), and so on. While this simple observation shows that the representation of the contextual information about our modeled systems is still very limited, it also shows that the framework, as we built it, could be applied – and offer added value – to systems development in other application areas.

5.1.1 Adequate purpose and scope

According to R1.1, our ontologies *"shall have an adequate purpose and scope"*, where *adequate* means *"sufficient for a specific requirement"* [68]. This requirement, in our case, is the expression of certain aspects of the Mercator TCS in the context of this PhD thesis. Even though multiple persons were involved in the TCS development, and these persons have "used" the ontologies indirectly via OntoManager (see 5.3.4), the ontologies were created by a single person and hence represent a very personal view on the world. The purpose and scope of the ontologies are therefore much different from those of the "informal ontologies" behind languages such as SysML and OPC UA, which are created in consensus by large organizations and are designed for a broad application range.

Looking back at the ontology descriptions in 3.3, it appears that the more abstract an ontology is, the more descriptive text is needed to justify the purpose and semantics of its concepts. The choices of what to represent (the scope), and why something should be represented (the purpose), are much easier to make for very specific domains of interest (such as IEC 61131-3 software development). Contradictorily, our most abstract ontologies – which are reused the most within the framework – may be the most "personal" and thus least reusable ontologies for external parties. The inverse relationship between the adequacy of ontologies for a specific project, and the number of stakeholders that needed to agree on the definition of the ontologies, also appears to hold for languages such as UML and SysML. These languages may not have the best scope and purpose when applied to specific projects because, as elaborated in 1.4, UML modelers typically ignore any UML semantics and invent their own [11] (indicating that UML is not used according to the intended purpose) and UML is considered unwieldy (indicating a problem of scope). It is therefore questionable whether or not a set of ontologies can have an adequate purpose and scope for a specific project, and still be agreed upon by a large community. Or, in other words, whether efforts should concentrate on finding a single set of ontologies that can act as a "lingua franca" for both internal and external stakeholders, or on trying to align two sets of ontologies with a very different scope and purpose.

The scope of the abstract ontologies determines, for instance, if the definition of the class `Requirement` should be part of the *development* ontology (as in 3.3.13), or of a smaller *requirements* ontology. The latter could be an example of a *fine grained ontology*, which is more reusable than a coarse grained ontology since it adheres more to the beneficial "single responsibility principle" of object-oriented design, and therefore it can more easily be "cherry picked" by modelers if needed. Of course, it does not mean that ontologies should be as small as possible (i.e. as in the extreme case of having one definition per ontology). There is sufficient value in combining strongly related concepts in a single ontology, since it allows us to refer to these concepts as a whole. For instance, the `owl:imports` relationship can be used to state that all definitions of the imported ontology are valid. While the `owl:imports` statement may lead

to similar problems as the *import* or *include* statements of object-oriented programming (e.g. "import moduleX" statements are less evolvable than "from moduleX import featureY" in Python), it is a convenient language feature if (and only if!) the imported ontologies are sufficiently fine grained. Looking at the list of the created ontologies (see table 3.1), we argue that our ontologies are fairly fine grained, although sometimes perhaps not sufficiently – as in the case of the *development* ontology.

A final observation is that, of the available third-party ontologies, we only reused the *QUDT* ontology. The *FOAF* ontology was also reused (to a much lesser extent), but in retrospect it might have been better to avoid this ontology since it defines fairly abstract concepts (such as a *document* and *person*) in an informal way – which opposes our goal to make formal ontologies. More formal third-party ontologies, such as *QUDT*, can be found on-line. Although some of them can be reused in our framework (for instance, the *provenance*[1] ontology also defines *persons* and *organizations*), we found that most third-party ontologies are difficult to integrate. Firstly, if third-party ontologies are very abstract, then they represent a "personal" view on the world (as explained above), which is likely inconsistent with our set of abstract ontologies – i.e. with *our* view on the world. More specific ontologies (e.g. one that represents electric components or one that represents the IEC 61131-3 concepts) would be much easier to reuse, but we could not find suitable ones (i.e. with a suitable purpose and scope) by searching the internet. Perhaps, this illustrates why we only reused *QUDT* to a large extent, since this ontology is only partially very abstract (as it defines *quantities* and *values*), but it mostly defines very specific and generally agreed upon concepts (such as the units of the *SI* system of measurement).

5.1.2 Rigorous formality

According to R1.2, our ontologies *"shall have a rigorous formality"*. Looking back at the code fragments in 3.3, it can be seen that no informal (English) comments have been added to the concept definitions. In theory, it means that our ontologies are indeed rigorously formal.

In practice however, much of the intended semantics of the ontology terms could not be expressed formally, so these semantics were simply left out. Partially, this was needed due to the limited expressiveness of the chosen knowledge representation languages. We think that the main reason, however, is that the scope and purpose of our ontologies is too vast to easily define all terms in a formal way. It would be a very daunting task to constrain the definitions of all terms of our ontologies, in a way that computers can fully grasp the intended meaning of these terms. Several terms (such as `mod:represents`, `ctrl:produces`, `dev:Project`, ...) proved to be too difficult to formally define. Hence, their correct usage depends on informal descriptions such as their English name, and their explanation in this thesis. "Correct" usage means that it is

[1] URI of the *provenance* ontology: `http://www.w3.org/ns/prov`.

in agreement with the intended meaning by the author (a single person) of the ontologies, which may be very personal as mentioned earlier in 5.1.1. A particular example of this is `sys:realization`, which was constrained in 3.3.1 by the rule saying that all properties and parts of the realization must be traceable to an element of the realized system. One might argue that this constraint is not strong enough: perhaps all semantic relations between the elements of the realized system must be present between the corresponding elements of the realizing system. SPIN can be used to express this: see listing 5.1. However, we did not implement this rule since it increases the cost of reasoning (it leads to almost three times more statements) without creating significant value (since most realizations in our models are generated by macros, that can also produce additional relations between the elements of the realizations, if needed). As will be argued in 6.3, a definition as the one of listing 5.1 could be applied however if the framework would be revised and some of the issues raised in this chapter would be addressed.

```
1  Class: Realization
2    EquivalentTo: realizes some System
3    SpinRule:
4      CONSTRUCT {
5          ?thisElement1   ?p              ?thisElement2
6      }
7      WHERE {
8          ?this           sys:realizes    ?otherSystem   .
9          ?this           sys:hasElement  ?thisElement1  .
10         ?this           sys:hasElement  ?thisElement2  .
11         ?thisElement1   sys:realizes    ?otherElement1 .
12         ?thisElement2   sys:realizes    ?otherElement2 .
13         ?otherSystem    sys:hasElement  ?otherElement1 .
14         ?otherSystem    sys:hasElement  ?otherElement2 .
15         ?otherElement1  ?p              ?otherElement2
16     }
```

Listing 5.1: Alternative definition of `sys:Realization`.

Due to the lack of constraints of the ontology terms, many non-sensical statements can be expressed that are not penalized by the semantic rules of the ontologies. For example, the statements `:m rdf:type elec:Motor` and `:m rdf:type iec61131:FunctionBlock` make no sense since an electric motor and an IEC 61131-3 function block are two disjoint concepts, yet a reasoner will not consider this model to be inconsistent since the disjointness of the concepts is not explicitly expressed. We mitigated this problem to a large extent by creating Ontoscript macros at the modeling level. The resulting frame-based DSLs (as shown in the example of listing 3.68) limit the freedom of the modeler, and therefore make it more unlikely that non-sensical statements are expressed unintentionally.

5.1.3 High extensibility

While some of our ontology terms are underconstrained, we think that many others are "correctly" constrained (i.e. as intended). When discussing the requirement of our ontologies to have a high extensibility (see 2.2.1, requirement R1.3) we called this the *right ontological commitment*, as in [9]. A good example is the definition of `mech:Motor`, which captures the most fundamental characteristic of a mechanical motor – as the equivalent of "something that produces mechanical power". Definitions like these are very extensible: the class `mech:Motor` can contain electric rotary or linear motors, but also pneumatic cylinders, ... or even human beings when we assert that they produce mechanical power. Whether or not "something that produces mechanical power" is rightfully called a "motor" is irrelevant to a reasoner, as a reasoner only cares about the formal semantics of a concept, and not about the semantics of its English name.

The problem of having overconstrained definitions (as in the case of SysML) is less likely when semantics are formally specified. From our experience of defining mathematical formulas (e.g. as in the *geometry* ontology where we attempted to express a dot product of two vectors, or as in the *mechanics* ontology where we expressed the transmission ratio as a simple division) it appears that the limited expressiveness of our chosen knowledge representation languages may also hamper extensibility. Expressing a complex formula using OWL, SPIN, and the concepts of our *expressions* and *mathematics* ontologies, appears to result in very extensive definitions (e.g. see listing 3.37). Supporting the expression of formulas in a more concise way using symbols (as in UML's OCL or SPARQL's FILTER expressions) may make the current ontologies, in practice, more extensible.

5.1.4 High clarity, concision, coherence and verifiability.

Looking back at the code listings in 3.3, it seems that high clarity is mostly achieved when *complete definitions* are given. Complete definitions are given when the `EquivalentTo` slots of class descriptions are filled out – representing the equivalency (the *if and only if* relationship) of two sets. The complete definition of `mech:Motor` (as mentioned above) is an example of this. High clarity is not only about formal semantics, however: it should be possible to intuitively associate the name of an ontology term with its formal meaning. For very specific concepts (which typically do not have a complete formal definition, as in the case of `iec61131:FunctionBlock`) this name should match closely the name of the concept which it represents. On the other hand, the definition of very abstract concepts (such as `dev:Constraint`) may be complete but also much more subjective, resulting in low clarity for other stakeholders.

In 2.2.1 we mentioned that a concise ontology only defines relevant terms, in agreement with the intended purpose and scope of the ontology. In retrospect, most of our "specific" ontology terms have been used directly to model systems (via Ontoscript), while most of our "abstract" ontology terms are reused by

multiple specific terms – indicating that they do factor out some commonality. Notable exceptions are the terms of the *geometry* ontology and the terms about temporal logic of the *expressions* ontology, as it turned out to be too difficult to demonstrate their added value. Other "problematic" ontology terms are the *has⌣* relationships, where ⌣ stands for the name of a class. For instance, `soft:hasVariable` is defined as a subproperty of `sys:hasPart`, with a range of `soft:Variable`. Very little expressive power is added to an ontology when a *has⌣* relationship is defined for each concept, while at the same it makes the ontology much less concise. The reason why we added several *has⌣* relationships is that the added expressive power may be little, but also very useful when constructing queries. For instance, matching the pattern `?x soft:hasVariable ?y` is much less costly than matching the pattern `?x sys:hasPart ?y . ?y rdf:type soft:Variable` (where '.' stands for logical *and*). This is because in the second pattern, evaluation of the first subclause (before the '.') may result in thousands of individuals due to the transitivity of `sys:hasPart`, and all of these individuals have to be matched against the second subclause (after the '.').

When ontologies only contain satisfiable concepts, they are called *coherent*. Coherence can be tested in practice: if an OWL reasoner infers that a class is a subclass of `owl:Nothing`, then this class is unsatisfiable. If expressiveness beyond OWL is used to define a class (e.g. as in the many cases where SPIN rules are expressed), then the reasoner should operate on a "test model", containing an individual of that class. On several occasions during the ontology development phase we have constructed such test models, assuring us that our ontologies are coherent. Of course, in our framework, all ontologies (except the *qudt* and *foaf* ontologies) have been created by a single person, which clearly makes it easier to achieve coherence.

The ability to formally verify system models on a semantic level – which is impossible to achieve with informal languages such as SysML – was one of the main drivers of our framework. Many constraints are expressed in our framework, as OWL 2 RL rules (to check cardinality constraints, to check if an instance is shared by two disjoint classes, ...) or as custom SPIN rules (to check if an expression has multiple different values at reasoning time, to check if the lefthand and righthand values of an equality are indeed equal, and so on). Most of these constraint rules operate on abstract concepts, making them very reusable within the framework. For instance, the formerly mentioned OWL 2 RL "disjointness rule" is both used to verify if two joined electric connectors have a compatible gender, but also to verify if the variables of an IEC 61131-3 function block are properly organized as input/output/inout/local/method members. Since system modelers may reuse these abstract concepts freely to express constraints at the system modeling level, the verification rules are applied (i) across the boundaries of engineering disciplines and (ii) across the boundaries of technologies. An example of (i) would be the expression that a transmission ratio of the mechanical model must equal the value of a constant software variable. An example of (ii) would be the expression that two software variables must have the same value, even if one is an IEC 61131-3 function block variable and the other is a Python class variable. Formal verification

within our framework is thus mainly achieved by constraint rules operating on abstract concepts. However, in practice, we feel that the ability to express formal constraints is currently much underused in our framework. Very few "real world problems" have been detected by the current framework implementation, during the development of the Mercator TCS. When modeling an electric cabinet consisting of a dozen connectors and several hundreds of terminals and wires, the chance that two male connectors are joined is very small compared to the chance that a wire is connected to a terminal having the wrong number. Verifying real world problems may therefore require the specification of more detailed concepts – having more detailed constraint rules. For instance, if our electric ontology would define categories of signals (e.g. power, data, TTL, analog, encoder A, phase B, ...), some of which are disjoint from each other, then the detection of wrongly specified connections would become much more likely.

5.2 DSLs

According to R2, *"DSLs shall be available to model the actual telescope control systems."* In our framework, these DSLs are provided by Ontoscript: an internal (embedded) DSL that can be used to map the concepts and properties of our framework's ontologies to Ontoscript primitives. For each ontology described in 3.3, a corresponding metamodel has been created in Ontoscript. These metamodels provide the primitives to construct the models of our actual systems, by defining individuals and by expressing facts about those individuals. Each metamodel can thus be regarded as a DSL, of which the semantics are defined by the corresponding ontology, and of which the syntax is defined by Ontoscript.

5.2.1 Textual notation

To satisfy R2.1, Ontoscript is a textual language, and therefore it avoids many of the pitfalls of graphical modeling languages. The problem of adding "unwanted" semantics is therefore less likely, but it is still present. The most prominent example of this is the support of Ontoscript for nested definitions, as shown in example code of listing 3.68. In this example, several requirements are defined within the concept definition, resulting in names such as `concept.rFoc`, `concept.rNasA` and `concept.rNasB`. The '.' in this name has no formal meaning: it is unrelated to the `sys:hasProperty` relations that were added explicitly by the macro of listing 3.67. Ontoscript allows us to create syntactically structured models without any formal semantic relation between two levels – the '.' thus only defines a namespace. Nesting makes the textual model descriptions more visually attractive (e.g. due to indentation), but it may cause confusion at the same time since it introduces no formal "semantic" dependencies. The "syntactic" dependencies that it introduces make the nested definitions well

organized but little evolvable at the same time, much like in object-oriented programming. For instance, if the name of the concept of listing 3.68 changes from `concept` to `baseConcept`, then all references to the concept's requirements have to be changed too (since the name of the requirements includes the name of the concept).

5.2.2 Rigorous syntax rules

Ontoscript is an internal DSL, and therefore its host language (CoffeeScript) defines the "minimal" syntactic rules that should be respected. These syntactic rules are formally defined by a grammar[2]. Evidently, not all CoffeeScript expressions are syntactically valid Ontoscript expressions. For instance, the expression:

```
mymodel.ADD sys.System "s"
```

is both a syntactically valid Ontoscript and CoffeeScript expression, while:

```
mymodel.ADD 3
```

is syntactically valid in CoffeeScript but not in Ontoscript.

Initially, an attempt was made to define a formal grammar for Ontoscript, to allow syntax verification before execution of the scripts. However, as explained in 3.4.1, one of the advantages of using an internal DSL is that also the expressiveness of the host language is available, if needed. Ontoscript is therefore not used as a traditional DSL, where a restricted set of host language features is used in a particular style. Effectively, we use the host language primitives as another "metamodel", e.g. to express variability, as in the following code:

```
mymodel.ADD sys.System "s#{i}" for i in [1..10]
```

As a consequence, the syntax of Ontoscript is little more restricted than the syntax of CoffeeScript. In practice, most syntax errors that we encountered during the development of the Mercator TCS were syntax errors of CoffeeScript, and were therefore immediately detected by the on-the-fly syntax verification service of our text editor. Other "Ontoscript-only" syntax rules are verified by the built-in checks of Ontoscript (for instance, the `ADD` function will raise an exception if an invalid argument is given). These errors are detected when the Ontoscript scripts are executed, much like many typical syntactic C++ errors are detected by the C++ compiler during the compilation step (and not on-the-fly by the programming editor). Because of these reasons, we did not create a formal grammar for Ontoscript.

5.2.3 High clarity and concision

According to R2.3, our *"DSLs shall have high clarity and concision"*. Comparing a very basic Ontoscript model (as in listing 3.63) with a more advanced

[2]See `http://coffeescript.org/v1/annotated-source/grammar.html`.

Ontoscript model that depends on macro expansion (as in listing 3.68), it is obvious that clarity is much related to the way in which Ontoscript is used. As elaborated in 3.4.1, Ontoscript macros allow us to efficiently create new "frame-based" DSLs, to efficiently deal with instantiation, and to simplify n-ary relations and other complex expressions as in the example of listing 5.2.

```
34    TRANSITION(
35        from      : $.states.opening
36        to        : $.states.aborted
37        within    : SEC(3)
38        condition : AND($.doAbort,
39                        LT(ABS(SUB($.actPos, $.targetPos)), $.posWindow))
40        ) "from_opening_to_aborted"
```

Listing 5.2: Example of expressing an n-ary relation.

The condition of the transition of listing 5.2 is expressed using macros (`AND`, `LT`, `ABS`, `SUB`) and is therefore fairly easy to understand. Clarity and concision could be improved however if the condition could be expressed in a symbolic language, as in the example of listing 5.3.

```
34    TRANSITION(
35        from      : $.states.opening
36        to        : $.states.aborted
37        within    : SEC(3)
38        condition : "$.doAbort && |$.actPos - $.targetPos| < $.posWindow"
39        ) "from_opening_to_aborted"
```

Listing 5.3: Improvement of listing 5.2.

Evaluation of such symbolic expressions is currently not supported, but it can fairly easily be implemented (in CoffeeScript) in the future.

Another currently missing feature of Ontoscript is a way to document macros. Clarity of the language would be much improved if a human readable list of all frames (e.g. `TRANSITION`) could be generated, and for each frame a reference to the associated ontology term (e.g. `fsm:Transition`), a list of valid slots for each frame (e.g. `from`, `to`, `within`, `condition`), and so on. Since such a feature is currently missing, modelers have to refer to the macro definitions to understand their usage. A solution is in preparation however: an *ontoscript* ontology is currently under development to represent macros, arguments, aliases, etc. This ontology can reuse some terms of the *software* ontology (such as `soft:Callable`, `soft:Argument`, and `soft:Assignment`). The idea is to modify the Ontoscript library so that a model that describes the macros is built in parallel to the models that are built while the Ontoscript scripts are being executed. A simple `WRITE` instruction could serialize this model (like any other model) at the end of the execution, and make it available to OntoManager for representation in the web-browser. Effectively, this would add so-called *introspection* functionality to Ontoscript.

5.3 Tools

According to R3, tools shall be available *"to support the knowledge-driven methodology applied to the development of telescope control systems."* The most prominent "supportive" tool of our methodology is OntoManager, since it provides the functionality to reason over the knowledge base, to query the knowledge base, and to generate artifacts. Other tools such as CoffeeScript programming editors are valuable too, as will be shown in the next subsection.

5.3.1 DSL syntax verification and mapping

In 2.2.3 we argued that tools should provide an easy way to verify the syntax of the DSL expressions, and an easy way *"to map the DSL expressions to expressions written in the knowledge representation language of the ontology behind the facade."* Since the Ontoscript syntax is not much more restricted than the CoffeeScript syntax (see 5.2.2), it means that any tool that supports CoffeeScript syntax verification is able to detect most syntactic errors of our Ontoscript models. Several popular text editors support CoffeeScript syntax coloring, and will notify the modeler "on-the-fly" when a syntactically invalid expression is written. OntoManager also includes a web-based editor that supports CoffeeScript, although currently editing scripts is not enabled and models can thus only be read.

To convert the Ontoscript models into RDF, we rely on the execution of the models by a JavaScript runtime platform (Node.js). OntoManager provides a web page (the *Dataset* page, described in 3.5.1) to control the execution of one or more models with just a few mouse clicks. One issue is that some syntactic or semantic errors raised during the execution of the models, may be difficult to be interpreted by a domain expert. If errors are not caught by the Ontoscript library, they will eventually be caught by the JavaScript runtime platform. Since such a "generic" JavaScript error cannot easily be traced back to an error in the Ontoscript domain-specific model, domain experts may find it difficult to diagnose the error description correctly. This problem is exacerbated by our choice for CoffeeScript as the host language of Ontoscript, since a JavaScript exception does not always clearly reveal the error in the CoffeeScript code, let alone the error in the Ontoscript model.

5.3.2 Reasoning

In order to support reasoning (R3.2), we built an open-source rules engine (SPIN API) into our OntoManager tool. The reasoning application, which we wrote in Java, can be commanded conveniently via the *Dataset* web page.

In table 5.1, we have listed the results of reasoning over the models of the subsystems of the Mercator TCS. Reasoning was performed on a modern laptop,

with OntoManager and the reasoner running on one core of a quad-core i7-3720QM 2.60 GHz processor. The columns of the table show, from left to right, the input of the reasoner (*Input*), the number of OWL classes (*#C*), the number of OWL properties (*#P*), the number of OWL individuals (*#I*), the number of asserted facts (*#asserted*), the number of inferred facts (*#inferred*), and the time it took to do the reasoning (*Time*). Reasoning was first performed over the individual subsystem models, and then over the complete knowledge base (the Mercator TCS or MTCS model). Reasoning over the models (i.e. the systems, mechanical, electrical, and software models) of a single subsystem typically happens only during the weeks that a particular subsystem is developed. Reasoning over the whole KB typically happens when the top-level models of the TCS change, when multiple subsystems are changed simultaneously, or when changes in one subsystem may affect other subsystems.

Table 5.1: Metrics for the Mercator TCS knowledge base.

Reasoner input	*#C*	*#P*	*#I*	*#asserted*	*#inferred*	*Time*
Services	1110	328	14 824	92 770	325 146	15^m55^s
Hydraulics	1110	328	21 578	132 181	467 512	30^m4^s
Cover	1110	328	21 255	129 887	447 889	27^m31^s
Safety	1110	328	17 455	107 654	388 535	21^m40^s
Telemetry	1110	328	26 484	152 550	674 918	54^m9^s
M1	1110	328	30 635	174 229	782 529	$1^h9^m29^s$
M2	1110	328	48 953	267 611	1 317 689	$3^h15^m57^s$
M3	1110	328	26 011	150 240	658 494	56^m1^s
Axes	1110	328	30 533	177 788	692 316	$1^h2^m34^s$
MTCS	1110	328	84 408	468 187	2 059 021	$8^h26^m2^s$

The number of classes (#C) and properties (#P) of all subsystems are equal because they depend on the same set of ontologies (the metamodels). The ontologies correspond to a so-called TBox (consisting of "terminological" knowledge) since they contain all semantic rules to expand the ABox (assertions about individuals, the models of our framework). Separation of the KB into a TBox and ABox is beneficial since it allows us to optimize the reasoning. To infer all facts from the models, we can iteratively apply all rules of the TBox to the ABox until no new facts are inferred. Our tools can thus be optimized: a traditional tableau-based[3] reasoner can be used to process the TBox efficiently (e.g. to verify satisfiability, to generate the subsumption hierarchy), while a rules engine (SPIN API in our case) can be used to process the ABox more efficiently [66].

The number of asserted facts, and the time it takes for the reasoner to reason over them, is very much tied to the way how our framework is implemented and how it is used. For instance, whenever a realization of a system (a software class,

[3]Tableau algorithms attempt to prove the satisfiability of concepts by constructing a model (a tableau, a graph of individuals) for each concept, and by expanding this model by applying the inference rules until no further inferences are possible or until a contradiction is found.

an I/O module type, a system design, ...) is expressed, then the Ontoscript macros of our framework will create new individuals for all parts and sub-parts of this system, at the time when the model is executed. This means that the size of the knowledge base would almost be doubled, if we would express a second realization of the largest modeled systems. As shown by example in listing 3.69, the expansion of realizations into parts and sub-parts is essentially a very crude way to allow us to reference those parts and sub-parts. Instead of storing all parts and sub-parts in memory, a so-called "lazy loading" implementation could only store those parts and sub-parts in memory that are referenced, when they are referenced. Such modifications of the current implementation may drastically reduce the number of asserted facts, and therefore drastically improve reasoning performance. Another disadvantage of the current implementation is that the whole KB has to be re-processed by the reasoner even if only a single fact of a single subsystem is added or removed. Nevertheless, the technologies we selected (such as JSON-LD and RDFLib) allow us to store the models as named graphs, so an improved implementation could require only the affected named graph to be re-processed by the reasoner.

In its current "naive" implementation, however, our KB consists of thousands of individuals which are not explicitly referenced (and therefore not strictly needed in memory), resulting in reasoning times of several hours. Such long reasoning times are not practical when the system is being developed: the current implementation makes it impossible to perform a quick verification of the models after each small change. As a consequence, verification can only be performed at regular times (e.g. every night, as in an automatic "nightly build" of a large software project). Another consequence is that, ideally, the queries of the predefined templates should not depend on the reasoning step (i.e. on the "materialization" of the inferences). Instead of inferring and storing ("materializing") all inferred knowledge in memory, we can rewrite the queries so that they return the correct result even if only the asserted facts are present in the knowledge base. For instance, the SPARQL pattern:

```
?x sys:hasPart ?y
```

can be rewritten as:

```
?x (sys:hasPart|(^sys:isPartOf))* ?y
```

to take into account the transitivity of `sys:hasPart` (via the SPARQL *zero-or-more-occurrences-of* operator *) and the inverse relation between `sys:hasPart` and `sys:isPartOf` (via the SPARQL *inverse-of* operator ^). Since RDFLib does not support automatic query rewriting, we manually rewrote the queries needed by the templates, so that the reasoning step is not needed to generate the artifacts. Reasoning is thus only needed for verification of the models, and for querying the KB with "free" queries. Since the template queries were rewritten manually, the OntoManager code has become more costly to maintain, but generation of the artifacts is much faster since no reasoning must be performed for every change of the models. This leads us to the conclusion that reasoning is, of course, supported by our framework – but its added value is very much limited by a few design choices of the current implementation.

5.3.3 Artifact generation

R3.3 is about the ability of OntoManager to generate artifacts such as web pages and source code files. Just like verification, artifact generation is essentially an example of information reuse. Once information is expressed in our framework as Ontoscript models, then this information can be verified (by the reasoner), it can be used to create HTML pages (by a template engine and a set of templates), and it can be used to create XML PLCopen or Python code files (by the same template engine, but a different set of templates). In the case of the Mercator telescope, several electrical subsystems of the Mercator TCS were built using only the generated HTML pages, and both the Mercator TCS and OCS (Observatory Control System) software include a significant amount of generated code (54 633 SLOC[4] of PLCopen-compliant XML code, and 4 345 SLOC of Python code). Since both the electrical and software systems of the telescope work as expected, we can argue that the generated artifacts indeed accurately represent the models – and thus also the modeled systems (since *represents* is a transitive property).

5.3.4 Usability

As elaborated in 2.2.3, usability is about the appropriateness of the tools (OntoManager in our case) to the development of telescope control systems. More in detail, usability is about:

- effectiveness: is OntoManager the right tool to use?
- efficiency: how much resources are saved by using OntoManager?
- satisfaction: how happy are the end-users to use OntoManager?

The effectiveness of OntoManager is not easy to evaluate objectively, since there is no comparison application: no attempt has been made to model, verify and generate artifacts of the Mercator TCS using another (existing) tool. We think however that OntoManager, in combination with a CoffeeScript editor, is quite "powerful" compared to some heavy commercial CASE tools that support SysML and code generation. Despite the limited development effort of a single person within the constraints of a PhD project, our tools cover the most important functions of these CASE tools – such as support for syntax verification, model transformations (by Ontoscript macros), and artifact generation. Moreover, we found that some features of OntoManager (such as a user-friendly way of querying the models, of creating instances (realizations), of reasoning over the models, etc.) are absent in the CASE tools that we tested. So while parts of OntoManager may not be the most effective pieces of software (e.g. a high performance graph store may be more effective than RDFLib, a versatile graph query language may be more effective than SPARQL, etc.), we do feel that

[4]Source lines of code (SLOC), including blank lines and comments.

OntoManager fills a niche since no comparable alternatives are available. It leads us to the conclusion that OntoManager is indeed an effective tool for developing telescope control systems – at least of the size of the Mercator telescope. For larger telescope control systems, implementation choices (e.g. with respect to lazy loading and named graphs) would have to be revised in order to maintain or improve the current level of usability.

The lack of a comparison application makes it also difficult to evaluate the efficiency of our tools: how much resources were saved or lost during the development phase, and how much resources will be saved or lost during the maintenance phase? Certainly, the design and implementation of OntoManager has required a significant investment of time – time that otherwise could have been spent on creating informal documents, SysML models, and software code that is directly written in TwinCAT (instead of being modeled first and then converted into code). We estimate that about 18 man-months were spent on the development of the framework (i.e. the ontologies, Ontoscript and OntoManager), and another 14 months on the developement of the TCS. Efforts during the remaining time of the PhD were more focused on the research described in chapters 1 and 2, on the development of TCS dependencies (such as our UAF software), and on the development of the first versions of the control systems of the MAIA instrument and the M3 support. These first versions were the result of an attempt to build more "industrial" control systems (the original intent of our PhD research project) via a traditional model-driven approach (see [83]). The lessons that we learned from this attempt were very valuable, since they made us realize why a shift from traditional "model-driven" development towards "knowledge-driven" development is needed.

While the effort of building a framework is justifiable within the PhD project to create a proof-of-concept, we estimate that a significant fraction of the development costs have already been compensated in the course of 2015-2016, when the framework was applied to the Mercator TCS. Since the first subsystem (the telescope cover) has been fully commissioned, we needed to apply only few changes to the OntoManager tool, to the ontologies, to the most important Ontoscript macros, and to the OntoManager templates. The development of the remaining subsystems of the TCS therefore went fast: Ontoscript proved to be a productive language to specify electric and software designs, and OntoManager proved to be a productive tool to quickly communicate designs to external stakeholders or to convert the designs into executable code. Any modifications of the designs, sometimes requiring extensive "refactoring" of the models, were especially fast to apply since the most important design logic is concentrated in a concise set of Ontoscript models. This advantage came to the surface very clearly when the *telescope axes* subsystem was replaced in June 2016: despite the very "invasive" operations (the complete removal of the transputers, SUN computer, telescope motors, drives, absolute encoders, ...), only two nights of technical downtime were lost before the first targets could be successfully observed again with the new software, motors, drives, encoders, etc. Whether or not the time gained during the TCS development and commissioning can compensate for the time lost during the framework development is hard to estimate, but clearly

any future additions and modifications to the TCS will benefit from the efforts made. Since the total maintenance costs of the TCS during its expected lifetime will likely be more substantial than the initial development costs (see 1.2.2), we think that the achieved efficiency will still increase in the years to come – despite the non-negligible extra cost of maintaining the framework.

Satisfaction of the end-users of OntoManager is hard to assess since most of the TCS was developed by a single person, and therefore not many end-users of the framework exist. One end-user is the company that was hired to draw the electric schematics and build the electric panels and cabinets of the telescope. For the first subsystems we specified the electric systems manually via spreadsheets, but as soon as the necessary templates were in place, we could simply provide this company a link to our on-line OntoManager application. Communicating the designs in this way was very helpful since i) the company found the set of interconnected HTML pages more interactive and user-friendly than the spreadsheets, and ii) we could fix any design errors found by the company very quickly by changing the models and immediately re-generating the HTML pages. Other end-users of the framework are the local staff members of the Mercator Telescope project at La Palma, who can now read/write/monitor/... any variable of the PLC conveniently using a handful of lines of Python code (as in the example of listing 4.1). Much more control is given to the developers of the OCS, because the OPC UA interface between the PLC-based TCS and the Python-based OCS is vastly more extensive and more powerful (in terms of features and performance) compared to the original TCS. Other "end-users" of the framework are the modelers: the domain experts that have to encode their designs as Ontoscript models. Since all current Ontoscript models were written by a single person, not much can be said about the satisfaction of those modelers. We expect however that future additions to the TCS may be modeled by other persons without many problems since i) a large amount of diverse existing models is available that can serve as "example code", ii) most Ontoscript code is based on a handful of high-level domain-specific Ontoscript macros, requiring very little familiarity with the underlying ontologies, and iii) the Ontoscript models are verified at several levels (e.g. macro arguments are verified by the macros, Ontoscript syntax is verified by the Ontoscript library), so that errors can be corrected as quickly as possible using the most appropriate diagnostic information.

5.4 Summary

In this chapter, we evaluated our framework by going through the original framework requirements that were outlined in chapter 2, thereby identifying the most important issues related to each requirement. We first observed that while our list of ontologies is extensive, the most application-specific domains of interest (e.g. about astronomy and telescopes) are not covered – making the framework widely applicable but perhaps not as effective for TCS development as it could have been. The purpose and scope of the ontologies are well

matched against the intended application, which may be both a strength and a weakness, depending on the expected acceptance of the ontologies by external stakeholders. All ontologies are of rigorous formality, although the formal semantics of the ontologies are much restricted by the limited expressiveness of the chosen knowledge representation languages, and by the wide scope and purpose of the ontologies. We consider the ontologies extensible since most terms are either underconstrained or "correctly" constrained – although the limited expressiveness of RDFS/OWL/SPIN hampers this extensibility somewhat. We further argue that the ontologies have a reasonably high clarity, concision, and coherence, but their verifiability is restricted by the lack of useful semantic rules for the most specific ontologies.

As for DSLs, many are available since Ontoscript is able to map any ontology to a DSL. Ontoscript offers a textual notation, although we note that this does not fully solve the problem of adding "unwanted" semantics. The syntax of Ontoscript is little more restricted than its host language, Coffeescript. We considered this both an advantage (because it allows us to exploit the expressiveness of CoffeeScript) and a disadvantage (because we were not able to create a complete formal grammar for Ontoscript). The DSL descriptions are concise and are of high clarity, especially if macros are used to create new "frame-based" DSLs.

Finally, our tooling is responsible for DSL syntax verification and mapping as required, although syntax verification is complicated by the embedded nature of the Ontoscript DSL. A reasoner was built into OntoManager, providing a means to verify the models in a convenient way. A few small but very important implementation choices result in poor reasoning times however, and therefore the added value of reasoning to our framework is lower than what we anticipated. Artifact generation has proved to be very useful on the other hand. The usability of our tools is hard to assess, but we do consider OntoManager effective because it offers several features that are not (or only poorly) supported by existing CASE tools. We think that using the framework saves resources, especially on longer term, but clearly the development costs of the framework were relatively high for a 1 meter-class telescope. Lastly, end-user satisfaction is hard to measure, although a few end-users have already clearly benefited from the framework. We expect that domain experts will be able to encode their designs in Ontoscript without many problems in the future, due to the availability of a large existing code base, due to the limited amount of frequently used high-level macros, and due to the verification logic that has been implemented at several levels.

Chapter 6

Conclusion

Knowledge-driven development has been identified by this thesis as the key towards more reusable (and, hence, traceable, verifiable and evolvable) telescope control systems. As telescopes continue to grow in size and complexity, we expect that three fundamental problems of current practices (more specifically: informal knowledge representation, graphical notation and overuse of object-oriented design patterns) will have to be addressed to improve reusability in the future.

In chapter 2, we argued that these current practices lead to models that can be classified as *information*: structured and processed but otherwise meaningless data about the structure and behavior of the telescope control systems. By formalizing the semantics of the primitives that specify these models, we aimed to turn this information into knowledge. *Formal knowledge representation* thus plays a pivotal role in the shift from traditional model-driven development to knowledge-driven development. One could say that the driving force behind this shift is the quest to eliminate the "human factor" hidden inside the models, by using a framework based on formal ontologies, textual domain-specific languages, and a supportive tool chain consisting of a knowledge base and a reasoner. Using such a framework reduces ambiguous human interpretation, it replaces the hierarchical and layered tree structure of typical object-oriented design by a much less constrained graph structure, and it avoids much of the implicit informal semantics introduced by graphical notation.

Chapter 3 forms the bulk of this thesis: it describes our implementation of the aforementioned framework. We laid out the framework architecture, and elaborated the constituent parts of the framework such as the metametamodels, metamodels, models, and tooling. Most attention is given to the metamodels: the formal ontologies that set our framework apart from traditional UML/SysML-based frameworks. They are key to understand the meaning of the actual system models written in our Ontoscript language, and to understand the opportunities of reasoning and querying to systems development, as provided

by our OntoManager application.

The implemented framework has been applied to the Mercator Telescope control system, as we described in chapter 4. This chapter not only serves as a quick introduction to the specifics and intricacies of the Mercator TCS, but it also illustrates the wide range of domain knowledge that can be captured by using our framework. At the same time, the constraints of the Mercator Telescope project also explain why the modeling of the electric and software designs is better covered than, for instance, the modeling of the systems and mechanical designs.

Chapter 5 finally evaluates the implementation of chapter 3 against the initial requirements of the framework as defined in chapter 2. In retrospect, this evaluation shows that satisfaction of the initial requirements is not a boolean property (yes or no) – unlike the definition of *satisfies* according to our *development* ontology. For each requirement, we have listed at least one issue that makes the requirement only partially satisfied. Still, the evaluation also shows that our framework has provided significant (but hard to quantify) added value, and that some of the issues can be resolved without too much effort. As will be repeated in the *Contributions* section of this chapter, we think that the most important outcome of the evaluation is that the framework proposed by this thesis can be implemented to such a degree that it can add value to the development of a *real* telescope control system, despite the constraints of the PhD project, and despite the competition of the "traditional" off-the-shelf solutions.

6.1 Validation

Whereas in the evaluation chapter we referred back to the framework requirements of chapter 2, in this section we will refer back to the problem definition of the first chapter. The question is thus not about the capability of our implementation to satisfy the initial framework requirements, but about the capability of such a framework to improve the reusability and lower the costs of telescope control systems in general.

Whether or not our particular implementation of the framework has improved the reusability (and lowered the costs) of the Mercator TCS, was already briefly discussed in 5.3.4 when we evaluated the usability of the tools. We concluded that such an assessment is very difficult to make (since there is no comparison application), and that the cost of building the framework was significant (albeit justifiable due to the reduced costs of commissioning and maintenance) compared to the cost of building the TCS in a traditional way. Generalizing this conclusion is difficult since the current implementation does not scale to much larger systems – at least not of the scale of VLTs and ELTs. As elaborated in chapter 5, scalability is mostly limited by a few known "technical" issues of the current implementation of the framework. Fixing some of them (in particular, by implementing a "lazy loading" mechanism to enable referring

to a particular element of a realization without materializing *all* elements of the realization) would much improve the scalability, and would allow us to model much larger systems using the existing ontologies, DSLs, and tools. If larger systems would be built using the framework, then the relative costs of the framework would be lower, and the potential for reuse of design knowledge would be higher. A first requirement of a generally applicable framework is therefore a high scalability – and thus the absence of a few "naive" technical implementation choices such as those discussed in chapter 5.

We think however that the real challenge of building a generally applicable framework, capable of improving reusability and lowering costs, is not of technical nature. The formalization of domain knowledge into ontologies, as we experienced it, turned out to be a much more difficult exercise than the development of the technical parts of the framework (i.e. the Ontoscript DSL and the OntoManager tool). Even the most commonly used terms, with seemingly evident semantics about the most frequently encountered concepts, are surprisingly difficult to formally (and even informally) define. What is an *instance* (or a *realization*) really? What kind of relationship exists between a *requirement* and a *constraint* of a system (if they are any different, at all)? Does our abstract definition of a software *variable* (as "something that represents a memory location and has exactly one symbol") correspond to what most people consider as a variable? Can the software concepts of *declaration*, *definition* and *implementation* be defined in a way that they can be reused by ontologies about different programming languages? Do these concepts need to be represented at all, to express the knowledge that we want to make reusable for future usage?

As discussed in 5.1.4, answering the above questions was much facilitated in this thesis by the small scale of the Mercator Telescope, since a single person could decide all definitions of the ontologies. A generally applicable framework on the other hand may involve many more domain experts – because it may be used for a much wider range of applications, because it may be used for much larger projects, and because it may be used on much longer time scales (with more people and technologies coming and going). Reaching a consensus among these experts may turn out to be very difficult, and hence the ontologies may always be tailored to a particular organization or project, and to the people and technologies of a particular time. This is illustrated by UML and SysML, which are designed to be "generally applicable", but which in reality are tailored to particular applications by ignoring the semantics of the official specifications. Whether or not a framework such as the one we implemented can add sufficient value to a project, may therefore much depend on the constraints of the project itself. We think however that, in general, if a project is sufficiently large to justify the cost of the ontology development (or "tailoring"), and the project is controlled by a sufficiently small group of domain experts, then building a framework as the one we proposed may be well worth the effort.

The costs and difficulties of building such a framework, as discussed above, naturally have to be matched against the gains of using the framework. Which real-world problems can be solved using a "knowledge-driven" development

methodology, that cannot be solved (or that can only be solved less efficiently) using a traditional methodology? What should be expressed by the ontologies and the models of the framework, instead of being left to more specific tools (such as CAD programs)? What is the real value of "knowledge" compared to "information", in practice? To answer these questions, we can look back at the application of chapter 4, and list the gains of which we think that have the highest potential, compared to traditional frameworks.

1. **Formal verification.** While models written in informal modeling languages cannot be verified on a semantic level, the models of our framework have to adhere to the rules specified by the ontologies. Even though we did not succeed to specify rules with 100% coverage that result in "provably correct" designs, any rule that we added has further constrained the system, and has led to a higher verifiability and less chances of errors. An important lesson that we learned is that, in retrospect, we should have focused more on the real-world problems: e.g. how can we verify if the data sending signal ("TxD") of an encoder ends up at a data receiving terminal ("RxD") of an I/O module? Despite the lack of such very specific rules in our framework, we did implement many other rules – since any statement of the ontologies represents a rule. As an analogy, we think that these semantic rules can be as useful to systems modeling, as unit tests are to programming. The higher the coverage, the better. But, if 100% coverage cannot be achieved (easily), then the focus should be on implementing the rules (or the unit tests) that have the highest probability of detecting errors. This does not necessarily mean that the rules must be explicitized in the ontology itself – as in our attempts to detect non-mutually exclusive states of a `fsm:Status` (see 3.3.9). On the contrary: a state machine ontology should perhaps better be based on (and thus implicitly constrained by) the input of a specific logic solver, so that the resulting models can easily be verified by this "external" (optimized) logic solver instead of the reasoner.

2. **Information reuse**. Compared to models written in informal modeling languages, the information described by our models is much more interconnected by semantic relationships. This can largely be attributed to the ontology definitions, which can only be expressed via concepts that are explicitly represented by other ontologies (or by metametamodels). On the other hand, the definitions of an informal language can be expressed using any (possibly ambiguous) word of a dictionary. For instance, when SysML specifies that "a requirement specifies a capability or a condition", we can only assume that the condition is a boolean expression, because a *condition* is not further explicitized. In contrast, our definition of `dev:Requirement` is bound to the definition of `dev:Constraint`, which is bound to the definitions of `expr:Always` and `expr:Constant`. It means that our tooling can be simplified because it can treat (e.g. query and visualize) the constraints of a requirement in a similar way as, for instance, a constant software variable, because both concepts are explicitly

connected by semantic relations. As another example, "everything is a system" – informally stated in 3.1.1 – has been made very explicit by our ontologies: we can extract the *properties* and *parts* of any concept via the `sys:hasProperty` and `sys:hasPart` relationship, respectively. However, as we elaborated in chapter 5, only a limited fraction of these "interconnections" are actually used by the queries of our OntoManager tool, indicating that we do not fully exploit the potential of the ontologies.

3. **Tool independence**. While traditional frameworks are very dependent on a CASE tool (probably not *any* CASE tool, but a specific one), our framework is composed of a set of tools that can be replaced without much effort. We think that one of the main reasons is that in our framework there is a relatively high amount of value in the models, whereas in traditional frameworks there is more value in the tools. A "knowledge-driven" methodology focuses very much on the semantics of the models, and thus leaves little freedom for the tools to add tool-specific interpretations. A "model-driven" methodology on the other hand is based on informal modeling languages, and therefore has to treat the models in a tool-specific way. While this observation holds for most parts of our framework, it is clear that some parts of our framework (such as the SPIN reasoner and the SPARQL query engine) are very dependent on the chosen knowledge representation languages, and thus cannot be replaced easily.

Judging from the above examples, it appears that our framework succeeds in bringing some of the potential of a knowledge-driven development methodology to the surface, albeit to a limited extent. There is no single revolutionary "showcase example" that makes our framework stand out from the traditional ones. Rather, we feel that our framework stands out because it provides many small carefully (and often pragmatically) selected advantages over the traditional tools, making our designs a bit more verifiable, more reusable, and less dependent on specific tools. Without these advantages, we think that it would have been impossible to build a framework, in the course of a PhD project, capable of modeling systems to such a finishing degree that they can be correctly interpreted by an external company to build electric cabinets, or by a third-party programming environment to build software for a complete PLC-based telescope control system. As we've written in the preface in the very beginning of this thesis, we thus think that there is no "revolutionary" step taken by our framework, but rather a small series of very small steps – the combination of which makes a difference.

6.2 Contributions

We think that the main contribution of this thesis is that it proves that our proposed methodology change is feasible: it *is* possible to design some important aspects of a *real* telescope control system in a formal and hence reusable way.

As we learned many times when talking to other organizations during the course of the PhD, several projects could benefit from such a "formalization of design knowledge", and from a framework as the one that we have built. Nevertheless, most of these organizations appear to only put trust in the most popular off-the-shelf CASE tools and modeling languages – the very source of their struggle, since these languages are informal and therefore intrinsically unqualified for formalizing a design. If the most important capabilities of the traditional tools can be surpassed by a framework built within the tight constraints of a PhD project and a 1 meter class telescope, then a more dedicated effort for a larger project may be able to create even much more value.

A second contribution consists of the lessons that we have learned, and that we have described in chapter 5. The issues raised in this chapter were discovered as we developed and applied our particular framework, but we have no doubt that most of them have to be addressed by anyone who builds a similar framework.

A third contribution is the set of ontologies and the open-source software that we developed during the thesis. Although the ontologies formalize a subjective view on the world, they are available on-line and can be reused by other projects. At the very minimum, they can serve as a set of examples (including good ones to be followed and bad ones to be avoided) when developing an improved set of ontologies. The value of our open-source software is more easy to prove, as our UAF project is currently used by several companies and research facilities[1].

A final contribution is the Mercator TCS itself: a reliable, user-friendly, well documented, evolvable, efficient, compact, and modern control system. Quantifying these qualities is difficult, but since we completed the system in June 2016, only very little technical downtime has been reported, and feedback by astronomers has been mostly very positive. Since most of the electric and software designs have been modeled using Ontoscript, documentation is extensive and consistent (since it is mostly generated), and changes or extensions are easy to apply (since they can be applied at a high level of abstraction). Efficiency is most apparent by the reduction of overhead during the execution of a pointing command or a guiding correction, which makes the new TCS noticeably more responsive than the original one. With "compactness" we refer to the small size of the components of the system, and the many times when we were able to replace a hardware solution by a software solution. As far as we know, the Mercator Telescope is the only operational telescope in the world that is fully implemented on a soft-PLC and fully integrated into the observatory-wide control system by an OPC UA information model, thereby taking full advantage of the versatility and performance of modern industrial platforms and communication technologies.

[1]Known users of the project are Weber Maschinenbau GmbH and Meyn Food Processing Technology B.V. (to control machines, in production); CERN (ATLAS experiment, where our UAF software plays a "central role" according to its users for testing general-purpose I/O boards or so-called ELMBs: Embedded Local Monitor Boards); the future CTA or Cerenkov Telescope Array (UI interfacing, and tests and calibration scripting for the NectarCAM instrument); and several others (e.g. see [75]).

6.3 Future prospects

A logical step to be taken from this point on, would be the improvement of the features that are currently already offered by our framework implementation. A consolidation of the current features – instead of the addition of new features – would at most require a few man-months, but it would much improve the usability of the framework. Efforts should focus on the most important issues identified in chapter 5. They may lead to improvements of Ontoscript, OntoManager (including new and improved templates), and likely a few small improvements of the ontologies.

A more challenging – but nevertheless realistic and rewarding – future research effort would be the complete revision of the framework, from the ground up. With the lessons learned from the current implementation, with additional research, and with less constraints imposed by the Mercator TCS, such a "framework 2.0" may look quite different from the first version created during the PhD project. A fresh look may lead to different knowledge representation languages, ontologies, system modeling languages, and tooling. More in-depth analysis of the current framework can reveal the expressiveness requirements of the knowledge representation languages, which may not be satisfiable by Semantic Web standards. Most effort would certainly have to be spent on the ontology development, which should focus on defining a very concise (but very well thought-through) set of abstract ontologies, serving as the foundation of the framework. A consensus among experts should be reached to define the semantics of the most important concepts such as systems, parts, instances (realizations), containers, expressions, and so on. Specific ontologies on the other hand should be as close to the "real world" as possible, and include semantic rules that solve real-world issues (as in the example of electric signal verification in 5.1.4). A revision of the framework may furthermore include a new "facade" for the ontologies: a revised Ontoscript, perhaps implemented as an external DSL to avoid the constraints of internal DSLs. Finally, there is much room for improving our tooling (OntoManager in particular), although we think that the concept of a multi-user web-based platform is likely to be maintained. Most revisions may take place "under the hood", perhaps by replacing the integrated graph store, query language, and reasoner. The result would be a very usable and generic framework, applicable to much larger and more complex systems than the Mercator Telescope, and much better prepared to adapt and extend these systems in the future compared to the traditional frameworks based on informal modeling.

Instead of revising the framework, effort could also be spent on implementing new features, to demonstrate some unprecedented opportunities. The models currently expressed in our framework are very "static": they only describe the design of the system. Without much effort, however, we could "link" this design information with real-time information of the actual system. Almost all information needed to accomplish this is already present in the knowledge base, as very detailed software models that are linked to electric and system

models. Real-time information of the Mercator TCS could easily be retrieved by OntoManager via OPC UA, since the information model of the OPC UA server of the PLC is fully available in the knowledge base. External companies have already been able to integrate our UAF software into Python web-server applications, so the integration of an OPC UA client into OntoManager would be straight-forward. If both "design-time" and real-time information are available, then several new applications can be thought of, including:

- web-based electric and software documentation with "live" status information, suitable for fast (cross-domain!) troubleshooting of technical problems;
- web pages that list the "live" unsatisfied requirements of the system (by continuously evaluating the modeled `dev:Constraint` expressions, either in OntoManager's web-server or directly in the JavaScript web pages);
- web pages that show a matrix of the "live" velocities of the parts of a gear train with respect to the other parts of the gear train;
- and so on...

If real-time data would be fed into the knowledge-base (which is not required for the above examples), then we could even query the knowledge-base with filters that operate on the real-time data. For instance, one could effectively query the knowledge-base to find all electric devices that are currently failing, and list their brand name and ID. Technical issues of data consistency and query performance may limit these opportunities, however. The other way around, some design data of the knowledge base could be inserted in the PLC via OPC UA (either via pushing or pulling). In this way, some data that is currently hard-coded in the generated source code, could be retrieved directly from the knowledge base of OntoManager. It would make the PLC software more flexible, and thus more evolvable.

Finally, the most certain future prospect is that the Mercator Telescope will continue to be operated and deliver scientific results in the foreseeable years to come. This will require regular maintenance of both software and hardware, and of both the framework and the TCS itself. As for the software, apart from fixing issues at the time when they occur, a regular (yearly or biyearly) maintenance should be planned. Ideally, all software and its dependencies (of both the framework and the PLC) should be updated, recompiled, redeployed and tested at regular intervals. Doing so will not prevent the dependencies of the system (such as OWL, SPARQL, EtherCAT, OPC UA, Python 2.7, TwinCAT 3, ...) from becoming obsolete at some point, but at least it will simplify migration, and detect the need for migration at the earliest time possible.

Appendix A

Appendix: Mercator TCS Ontoscript examples

In this appendix, a number of models of the Mercator TCS are shown to illustrate the usage of Ontoscript. The shown models represent the *telescope cover* subsystem, or parts of it. This subsystem has been documented in 4.3.5: it consists of two sets of four aluminum petals that open or close, like the petals of a flower. We chose this subsystem because we developed it from scratch during the PhD project, whereas most other subsystems were only equipped with new electronics and software. Only a fraction of the models of the telescope cover are displayed, since the complete models are too extensive to be included in this appendix. In the sections below we will elaborate the models about the systems engineering (`cover_sys`), mechanical engineering (`cover_mech`), electric engineering (`cover_elec`), and software engineering (`cover_soft`) aspects of the telescope cover, respectively.

A.1 Systems engineering

We created a "systems engineering" model with prefix `cover_sys` and with URI `http://www.mercator.iac.es/onto/models/mtcs/cover/system` to describe the concept of the telescope cover, and to describe the various system designs (from the top level to a more detailed level) that realize this concept. The other models (i.e. those that describe the mechanical, electrical and software engineering aspects of the telescope cover) all import this model to be able to refer to it, as will be shown in the remaining sections of this appendix. Listing A.1 shows how the `cover_sys` model is defined, and how a project (a `dev:Project` instance) is added to the model. Any concepts or designs modeled afterwards will be linked to this project (via the `sys:isPartOf` relationship).

```
##############################################################################
#                                                                            #
# Systems engineering of the telescope cover.                                #
#                                                                            #
##############################################################################

require "ontoscript"

# require the dependencies
REQUIRE "models/import_all.coffee"
REQUIRE "models/util/systemfactories.coffee"
REQUIRE "models/mtcs/common/roles.coffee"

MODEL "http://www.mercator.iac.es/onto/models/mtcs/cover/system" :
        "cover_sys"

# import the dependencies
cover_sys.IMPORT systemfactories
cover_sys.IMPORT roles

##############################################################################
# The project
##############################################################################

cover_sys.ADD MTCS_PROJECT "project",
    label: "Cover"
    comment: "The cover of the telescope"
```

Listing A.1: Systems engineering model example.

We can now define a concept of the telescope cover, and add it to the project. Concepts can be described by calling the `MTCS_CONCEPT` macro. This macro creates a `dev:Concept` instance: a "black box" model that does not have any parts (i.e. a concept can only be described as a whole, see 3.3.13). As can be seen in listing A.2, macro arguments (or frame "slots") are available to describe the requirements of the concept, its properties, the states and transitions of its state machine behavior, its constraints that must be asserted, and some tests to verify the requirements that cannot easily be verified by subsequent modeling. Concept descriptions speak in very general terms about an envisioned system, they should not restrict any future designs unnecessarily.

```
##############################################################################
# TC: Concept
##############################################################################

cover_sys.ADD MTCS_CONCEPT "concept",
  comment: "Concept of the Mercator telescope cover"
  partOf: cover_sys.project
  requirements:
    covered: ->
      comment: "Have a 'covered' state, protecting the tube against " +
               "small hazardous parts and falling water drops (IP32)"
      isRequiredBy: roles.tech
    uncovered: ->
      comment: "Have an 'uncovered' state which doesn't obstruct the beam"
      isRequiredBy: roles.observer
    heat: ->
      comment: "Dissipate max. 30 Watts inside the dome, when uncovered"
      isRequiredBy: roles.observer
    emc: ->
      comment: "Don't produce electric noise that can interfere with " +
```

```
                "observations"
      isRequiredBy: roles.observer
    switching: ->
      comment: "Be able to switch between covered/uncovered"
      isRequiredBy: roles.observer
    covering: ->
      comment: "Be able to switch to covered state within 120s"
      isDerivedFrom: self.switching
    uncovering: ->
      comment: "Be able to switch to uncovered state within 120s"
      isDerivedFrom: self.switching
    automated: ->
      comment: "Be able to open/close remotely with a single command"
      isRequiredBy: roles.observer
    safe: ->
      comment: "Be intrinsically safe"
      isRequiredBy: [roles.observer, roles.tech]
    wind: ->
      comment: "Be resilient to 'high' wind load while (un)covered."
      isRequiredBy: [roles.observer, roles.tech]
  properties:
    maxSwitchingTime   : ->
        comment : "Maximum switching time"
        value   : 120
        unit    : unit.SecondTime
    maxHeatDissipation : ->
        comment : "Maximum heat dissipation"
        value   : 30
        unit    : unit.Watt
    actHeatDissipation : ->
        comment : "Actual heat dissipation"
        unit    : unit.Watt
  states:
    # the cover has covered/uncovered states
    covered    : -> comment: "Fully covered"
    uncovered  : -> comment: "Fully uncovered"
    # the cover also has covering/uncovering states
    covering   : -> comment: "Switching to covered state"
    uncovering : -> comment: "Switching to uncovered state"
  constraints:
    uncovering: ->
      always:
         if   : $.states.uncovering
         then : EVENTUALLY($.states.uncovered,
                           within: $.properties.maxSwitchingTime)
      represents: $.requirements.uncovering
    covering: ->
      always:
         if   : $.states.covering
         then : EVENTUALLY($.states.covered,
                           within: $.properties.maxSwitchingTime)
      represents: $.requirements.covering
    heat: ->
      always:
        if   : $.states.uncovered
        then : LT($.properties.actHeatDissipation,
                  $.properties.maxHeatDissipation)
      represents: $.requirements.heat
  tests:
    covered: ->
      comment   : "Test if the covered state fully covers the tube, by
          visual inspection"
      verifies  : $.requirements.covered
    uncovered: ->
      comment   : "Test if the uncovered state doesn't obstruct the beam,
          by visual inspection"
      verifies  : $.requirements.uncovered
```

Listing A.2: Systems engineering model example (continued).

The concept described above can be realized by a system design (or multiple even, for trade-off studies). System designs can be described using the `MTCS_DESIGN` macro, which produces a `dev:Design` instance. As can be seen in listing A.3, this macro has two additional slots compared to the `MTCS_CONCEPT` macro: `realizes` to link the design to the concept that it realizes (see line 151), and `parts` to break down the "black box" into its parts (see line 182). By realizing the previously described concept, the system design "inherits" the concept's requirements, properties, states, constraints, etc. Four parts have been defined: a set of "bottom" panels, a set of "top" panels (which overlap with the bottom panels), an electric cabinet, and the software of the system. These parts are instances of `dev:Concept` and must be realized by subsequent `dev:Design` instances. Thus, every time we use the `MTCS_DESIGN` macro, we realize another concept and we break down the system into more parts, until all parts have been realized. Since the two panel sets (top and bottom) are very similar, we could defined parametric function (`createPanelSetConcept`, lines 122–144) to describe their commonalities.

```
113 ##############################################################################
114 # TC: System design
115 ##############################################################################
116
117 # Since the systemDesign will define two very similar "panel set"
118 # concepts, we can create a function to generate a panel set concept.
119 # This function will be called twice within the systemDesign definition
120 # (once to generate the "top" panel set concept, once to generate the
121 # "bottom" panel set concept).
122 createPanelSetConcept = (name) ->
123   comment    : "The #{name} panel set"
124   requirements:
125     open: ->
126       comment: "Have an open state"
127       isDerivedFrom: $.$.$.requirements.open
128     closed: ->
129       comment: "Have a closed state"
130       isDerivedFrom: $.$.$.requirements.closed
131     safe: ->
132       comment: "Be intrisically safe"
133       isDerivedFrom: $.$.$.requirements.safe
134     actuated: ->
135       comment: "Be actuated"
136       isDerivedFrom: $.$.$.requirements.automated
137     wind: ->
138       comment: "Be resilient to high wind load when open/closed"
139       isDerivedFrom: $.$.$.requirements.wind
140   states:
141     open   : -> {} # TBD by the realization
142     closed : -> {} # TBD by the realization
143   properties:
144     powerDuringObservations: -> {} # TBD by the realization
145
146 # Below we define the systemDesign.
147 # It is a realization of the previously defined system concept
148 # (cover_sys.concept).
149 cover_sys.ADD MTCS_DESIGN "systemDesign",
150   comment: "System design of the Mercator telescope cover"
151   realizes: cover_sys.concept
152   requirements:
153     # Requirements defined by the system concept were added automatically
154     # by realization (i.e. when the when the "realizes: ..." line
155     # was executed).
156     # The requirements below are new requirements, derived from the
```

```
157       # realized ones.
158       panelSets: ->
159         comment: "Have two sets of overlapping panels, to improve sealing"
160         isDerivedFrom: $.requirements.covering
161       open: ->
162         comment: "Be uncovered if both panel sets are open"
163         refines: $.requirements.uncovered
164       closed: ->
165         comment: "Be covered if both panel sets are closed"
166         refines: $.requirements.covered
167       opening: ->
168         comment: "Be opening on command within 3 seconds"
169         isDerivedFrom: [$.requirements.uncovering, $.requirements.automated]
          # remaining requirements (closing, stopping, stopPriority,
          # closePriority) are not shown in this listing

182     parts:
183       # the top and bottom panelSet concepts:
184       top: -> createPanelSetConcept("top")
185       bottom: -> createPanelSetConcept("bottom")
186       # the electric cabinet concept, to be realized below:
187       cabinet: ->
188         comment    : "Electrical cabinet concept"
189         requirements:
190           safe: ->
191            comment: "The cabinet shall be safe"
192            isDerivedFrom: $.$.$.requirements.safe
193       # the software, to be realized by the cover_soft model:
194       software: ->
195         comment: "Control software concept"
196     properties:
197       heatDissipation: ->
198           sameAs  : SUM($.parts.top.properties.powerDuringObservations,
199                         $.parts.bottom.properties.powerDuringObservations)
200       accelerationTime: ->
201           comment : "Maximum time to accelerate"
202           unit    : unit.SecondTime
203           value   : 3.0
204       decelerationTime: ->
205           comment : "Maximum time to decelerate"
206           unit    : unit.SecondTime
207           value   : 3.0
208     states:
209       open   : -> sameAs: AND($.parts.top.states.open,
210                               $.parts.bottom.states.open)
211       closed : -> sameAs: AND($.parts.top.states.closed,
212                               $.parts.bottom.states.closed)
213       partiallyOpen: -> sameAs: NOT(OR($.states.open, $.states.closed))
214       # the cover can be opening, closing, or stopped
215       opening    : -> comment: "The cover is opening"
216       closing    : -> comment: "The cover is closing"
217       stopped    : -> comment: "The cover is stopped"
218       # three commands can control the cover
219       doClose    : -> comment: "The cover is commanded to close"
220       doOpen     : -> comment: "The cover is commanded to open"
221       doStop     : -> comment: "The cover is commanded to stop"
222     transitions:
223       stopped_to_opening: ->
224         from     : $.states.stopped
225         to       : $.states.opening
226         condition: AND($.states.doOpen,
227                        NOT(OR($.states.doClose,$.states.doStop)))
228         within   : $.properties.accelerationTime
229       stopped_to_closing: ->
230         from     : $.states.stopped
231         to       : $.states.closing
232         condition: AND($.states.doClose, NOT($.states.doStop))
233         within   : $.properties.accelerationTime
234       opening_to_stopped: ->
235         from     : $.states.opening
```

```
        to       : $.states.stopped
        condition: $.states.doStop
        within   : $.properties.decelerationTime
      closing_to_stopped: ->
        from     : $.states.closing
        to       : $.states.stopped
        condition: $.states.doStop
        within   : $.properties.decelerationTime
    constraints:
      opening: ->
        always     : $.transitions.stopped_to_opening
        represents : $.requirements.opening
      closing: ->
        always     : $.transitions.stopped_to_closing
        represents : [$.requirements.closing, $.requirements.closePriority]
      stop: ->
        always     : AND($.transitions.opening_to_stopped,
                         $.transitions.closing_to_stopped)
        represents : [$.requirements.stopping, $.requirements.stopPriority]
```

Listing A.3: Systems engineering model example (continued).

An example of a more detailed design is the `cover_sys:panelDesign` shown in listing A.4, which describes a single "panel" of the cover. A panel is a system that consists of several parts: a petal, a bracket, a motor, an encoder, ... As in the previous example, these parts are `dev:Concept` instances; they still have to be realized by more specific instances. The latter are not described in the `cover_sys` systems engineering model, but instead they are described in the mechanical and the electric engineering models (see next sections).

```
###############################################################################
# The panel design
###############################################################################

cover_sys.ADD MTCS_DESIGN "panelDesign",
  comment: "The design of the telescope cover panels"
  realizes: cover_sys.panelSetDesign.parts["p#{i}"] for i in [1..4]
  requirements:
    # Previously defined requirements (open, closed, safe, ...) at the
    # concept level have been added by realization.
    # The requirements below are derived from them.
    tiltingPetal: ->
        comment: "A petal can tilt around an axis, to an open or " +
                 "closed position"
        isDerivedFrom: [ $.requirements.open, $.requirements.closed ]
    clamping: ->
        comment: "A petal can be 'clamped' by pressing the flexible " +
                 "petal against the tube or M2 before releasing power"
        isDerivedFrom: self.tiltingPetal
    closedLoop: ->
        comment: "The panel is controlled in closed loop"
        isDerivedFrom: self.clamping
    absFeedback: ->
        comment: "Absolute position feedback is available"
        isDerivedFrom: self.closedLoop
    absFeedbackStatus: ->
        comment: "The status of the absolute feedback shall be known"
        isDerivedFrom: cover_sys.systemDesign.requirements.automated
  parts:
    # Below we define concepts. These concepts still have to be realized
    # (e.g. by the realizations in the cover_mech and cover_elec models).
    petal: ->
      requirements:
```

```
288           flexible: ->
289             comment: "Be made of flexible material, to allow clamping"
290             isDerivedFrom: [ $.$.$.requirements.clamping,
291                              $.$.$.requirements.tiltingPetal ]
292       shaft: ->
293         requirements:
294           fixedToPetal: ->
295             comment: "Petal shaft, mounted inside bearings of the bracket"
296             isDerivedFrom: $.$.$.requirements.tiltingPetal
297       magnet: ->
298         requirements:
299           power: ->
300             comment: "Sufficiently low to minimize heat dissipation, " +
301                      "sufficiently high to hold petal during wind gusts"
302             isDerivedFrom: [ cover_sys.systemDesign.requirements.heat
303                              $.$.$.requirements.wind ]
304         properties:
305           power: ->
306             comment: "Electric power consumption"
307         states:
308           on: -> comment: "Powered on"
309           off: -> comment: "Powered off"
      # remaining parts (bracket, motor, encoder, slipClutch, mot_to_shaft,
      # enc_to_shaft) are not shown in this listing

343   properties:
344     closedPosition  : ->
345         comment: "Closed position (value TBD by software)"
346         unit: unit.DegreeAngle
347     openPosition    : ->
348         comment: "Open position (value TBD by software)"
349         unit: unit.DegreeAngle
350     maxClampingTime : ->
351         comment: "Max. duration between the time when the panel is " +
352                  "open/closed, and the time when the panel is clamped " +
353                  "and the motor is turned off"
354         unit: unit.SecondTime
355         value: 5.0
      # remaining properties (actualPosition, openTolerance,
      # closedTolerance) are not shown in this listing

367   states:
368     open: ->
369       sameAs: LT( ABS( SUB($.properties.actualPosition,
370                            $.properties.openPosition  ) ),
371                   $.properties.openTolerance )
372     closed  : ->
373       sameAs: LT( ABS( SUB($.properties.actualPosition,
374                            $.properties.closedPosition) ),
375                   $.properties.closedTolerance )
376     clamped: ->
377       sameAs: AND( OR($.states.open, $.states.closed),
378                    $.parts.motor.states.off,
379                    $.parts.magnet.states.on)
380   constraints:
381     powerDuringObservations: ->
382       always:
383         if: AND($.states.open, $.states.clamped)
384         then: EQ($.properties.powerDuringObservations,
385                  $.parts.magnet.properties.power)
```

Listing A.4: Systems engineering model example (continued).

A.2 Mechanical engineering

As explained in 3.3.15, the models about the mechanics of the Mercator telescope are very limited (or often absent), since most of the mechanics of the telescope were left untouched by the new telescope control system. Listing A.5 illustrates how some particular custom-built parts of the cover have been described very briefly by the mechanics model (`cover_mech`), without detail. Naturally, they are described in great detail by CAD models, so it makes little sense for these parts to be described in more detail in a reusable way by our models. Still, one could see how some aspects of these parts (such as, perhaps, their material and their weight) could be useful reusable information. Extensions like these could be added to the framework in the future without much effort.

```
1  ##############################################################################
2  #                                                                            #
3  # Model of the telescope cover mechanics.                                    #
4  #                                                                            #
5  ##############################################################################
6
7  require "ontoscript"
8
9  # models
10 REQUIRE "models/mtcs/cover/system.coffee"
11 REQUIRE "models/external/maxon.coffee"
12 REQUIRE "models/external/kuebler.coffee"
13 REQUIRE "models/external/magnetschultz.coffee"
14
15 MODEL "http://www.mercator.iac.es/onto/models/mtcs/cover/mechanics" :
16         "cover_mech"
17
18 cover_mech.IMPORT cover_sys
19 cover_mech.IMPORT mech
20 cover_mech.IMPORT maxon
21 cover_mech.IMPORT kuebler
22 cover_mech.IMPORT magnetschultz
23
24 ##############################################################################
25 # Custom-built parts
26 ##############################################################################
27
28 cover_mech.ADD mech.PART(
29     comment  : "Aluminum petal, anodized",
30     realizes : cover_sys.panelDesign.parts.petal
31     ) "petal"
32
33 cover_mech.ADD mech.PART(
34     comment  : "Shaft of the petal",
35     realizes : cover_sys.panelDesign.parts.shaft,
36     fixedTo  : cover_mech.petal
37     ) "shaft"
38
39 cover_mech.ADD mech.PART(
40     comment  : "Bracket",
41     realizes : cover_sys.panelDesign.parts.bracket
42     ) "bracket"
```

Listing A.5: Mechanical engineering model example.

The off-the-shelf parts of the cover have been described more in detail, as shown in listing A.6. Not only do they realize the concepts of the `cover_sys`

model via the **realizes** slot, but they also specify the type of the part via the **type** slot (which links the part to its type via the **man:hasType** relationship). Since **man:hasType** is a subproperty of **sys:realizes**, the parts shown below effectively realize two systems.

```
43  ##############################################################################
44  # Motor, encoder and magnet
45  ##############################################################################
46
47  cover_mech.ADD ROTARY_MOTOR(
48      comment       : "Actuator of the petal",
49      realizes      : cover_sys.panelDesign.parts.motor,
50      type          : maxon.motor_370418,
51      stator: ->
52          fixedTo   : cover_mech.bracket
53      ) "motor"
54
55  cover_mech.ADD ROTARY_TRANSMISSION(
56      comment       : "Planetary reduction fixed onto the motor",
57      type          : maxon.reduction_166960
58      stator: ->
59          fixedTo   : cover_mech.motor
60      inputRotor: ->
61          fixedTo   : cover_mech.motor.rotor
62      ) "motorReduction"
63
64  cover_mech.ADD ROTARY_LOAD(
65      comment       : "External encoder",
66      realizes      : cover_sys.panelDesign.parts.encoder,
67      type          : kuebler.F3673_1421_G412,
68      stator: ->
69          fixedTo   : cover_mech.bracket
70      ) "encoder"
71
72  cover_mech.ADD mech.PART(
73      comment  : "Magnet",
74      realizes : cover_sys.panelDesign.parts.magnet,
75      type     : magnetschultz.G_MH_x025
76      ) "magnet"
```

Listing A.6: Mechanical engineering model example (continued).

The macros shown in listing A.6 produce much more facts than those expressed in Ontoscript, due to the realizations by the **realizes** and/or **type** slots. For instance, the **encoder** macro call of lines 64-70 will first create corresponding properties of the **cover_sys.panelDesign.parts.encoder** concept, and will then create corresponding parts and properties of the **kuebler.F3673_1421_G412** encoder type of the Kübler manufacturer. This encoder type has been described in a model about the products by Kübler that we use. A small excerpt of this model is shown in listing A.7.

```
33  kuebler.ADD CABLE_TYPE(
34    id              : "Cable9CoreScreen"
35    comment         : "Kubler SSI encoder cable"
36    manufacturer    : kuebler.company
37    wires:
38      white  : -> symbol: "WH", comment: "White wire" , color: colors.white
39      brown  : -> symbol: "BN", comment: "Brown wire" , color: colors.brown
40      green  : -> symbol: "GN", comment: "Green wire" , color: colors.green
```

```
    yellow : -> symbol: "YE", comment: "Yellow wire", color: colors.yellow
    gray   : -> symbol: "GY", comment: "Gray wire"  , color: colors.gray
    pink   : -> symbol: "PK", comment: "Pink wire"  , color: colors.pink
    blue   : -> symbol: "BU", comment: "Blue wire"  , color: colors.blue
    red    : -> symbol: "RD", comment: "Red wire"   , color: colors.red
    violet : -> symbol: "BK", comment: "Violet wire", color: colors.violet
    screen : -> symbol: "S" , comment: "Cable screen"
  ) "Cable9CoreScreen"

kuebler.ADD SENSOR_TYPE(
  id              : "F3673.1421.G412"
  comment         : "SSI encoder ST=14-bit"
  manufacturer    : kuebler.company
  cables:
    cable       : ->
      type: kuebler.Cable9CoreScreen
      comment: "Cable coming out of the encoder"
      wires:
        # all wires below are described by kuebler.Cable9CoreScreen !
        white   : -> comment: "0V"
        brown   : -> comment: "+V (10..30VDC)"
        green   : -> comment: "Clock +"
        yellow  : -> comment: "Clock -"
        gray    : -> comment: "Data +"
        pink    : -> comment: "Data -"
        blue    : -> comment: "SET (set zero or a predefined value)"
        red     : -> comment: "DIR (set counting direction)"
        violet  : -> comment: "STAT (status, +V=OK, 0V=Fault)"
        screen  : -> comment: "Shield"
  ) "F3673_1421_G412"
```

Listing A.7: Vendor-specific model model example.

The mechanical model finally describes some more transmissions, as shown in listing A.8. The missing transmission ratio of the `mot_to_shaft` transmission can be calculated because the ratios of the "intermediary" transmissions (`motorReduction` of listing A.6, and `red_to_clutch` and `slipClutch` of listing A.8) are all known.

```
###############################################################################
# Transmissions
###############################################################################

cover_mech.ADD ROTARY_TRANSMISSION(
    comment       : "Transmission from motor shaft to petal shaft",
    realizes      : cover_sys.panelDesign.parts.mot_to_shaft,
    stator: ->
        fixedTo   : cover_mech.bracket
    inputRotor: ->
        fixedTo   : cover_mech.motor.rotor
    outputRotor: ->
        fixedTo   : cover_mech.shaft
    ) "mot_to_shaft"

cover_mech.ADD ROTARY_TRANSMISSION(
    comment       : "Transmission from motor reduction to slip clutch",
    ratio         : 1/8
    stator: ->
        fixedTo   : cover_mech.bracket
    inputRotor: ->
        fixedTo   : cover_mech.motorReduction.outputRotor
    ) "red_to_clutch"

cover_mech.ADD ROTARY_TRANSMISSION(
```

```
    comment       : "Slip clutch, mounted between petal shaft and the " +
                    "gear driven by the motor",
    realizes      : cover_sys.panelDesign.parts.slipClutch,
    ratio         : 1/1
    inputRotor: ->
        fixedTo   : cover_mech.red_to_clutch.outputRotor
    outputRotor: ->
        fixedTo   : cover_mech.shaft
    ) "slipClutch"

cover_mech.ADD ROTARY_TRANSMISSION(
    comment       : "Transmission from encoder shaft to petal shaft",
    realizes      : cover_sys.panelDesign.parts.enc_to_shaft,
    ratio         : 1/1
    stator: ->
        fixedTo   : cover_mech.bracket
    inputRotor: ->
        fixedTo   : cover_mech.encoder.rotor
    outputRotor: ->
        fixedTo   : cover_mech.shaft
    ) "enc_to_shaft"
```

Listing A.8: Mechanical engineering model example (continued).

A.3 Electrical engineering

The `cover_elec` model describes the electric system of the telescope cover. This system is composed of instances (realizations) of device types (power supplies, I/O modules, motors, ...), and of cables, wires, connectors, power terminals, etc. We start by defining a configuration that will contain all electric devices.

```
##############################################################################
#                                                                            #
# Model of the telescope cover electric system.                              #
#                                                                            #
##############################################################################

require "ontoscript"

# models
REQUIRE "models/external/all.coffee"
REQUIRE "models/mtcs/cover/system.coffee"

MODEL "http://www.mercator.iac.es/onto/models/mtcs/cover/electricity" :
      "cover_elec"

cover_elec.IMPORT external_all
cover_elec.IMPORT cover_sys

##############################################################################
# Configuration
##############################################################################

cover_elec.ADD elec.Configuration "TC": [
  LABEL "TC: Telescope Cover"
  COMMENT "The dust cover of the telescope"
]
```

Listing A.9: Electrical engineering model example.

In listing A.10, we will first create an instance of a power input connector, connected to a circuit breaker. We will also describe a list of power terminals (only one per signal type), and a 24V DC power supply instance that is fed by the circuit breaker and feeds the power terminals. As with all devices of the `cover_elec` model, the type of each instance has been described earlier in a vendor-specific model (such as `phoenix` for Phoenix Contact products, `schneider` for Schneider Electric products, `beckhoff` for Beckhoff products, and so on). When the `CONNECTOR_INSTANCE` macro call is executed, an extensive *realization* is thus created, consisting of all terminals that can be realized by inspecting the connector type defined by the vendor-specific model. As advocated by this thesis, there is no abstraction or "encapsulation" of the specific details of the selected devices: we are very specific about them.

```
27  # Power input ============================================================
28
29  cover_elec.TC.ADD cont.contains CONNECTOR_INSTANCE(
30    type: phoenix.SC25_1L_SocketAssembly
31    comment: "230V input power"
32    symbol: "TC:230VAC-A"
33    ) "socket230VAC"
34
35
36  cover_elec.TC.ADD cont.contains CIRCUIT_BREAKER_INSTANCE(
37    type: schneider.CircuitBreaker2Ph6A
38    symbol: "TC:CB"
39    comment: "Circuit breaker immediately after 230V input"
40    terminals:
41      1: ->
42        comment: "L in"
43        isConnectedTo: cover_elec.TC.socket230VAC.terminals.L
44      2: ->
45        comment: "L out"
46      3: ->
47        comment: "N in"
48        isConnectedTo: cover_elec.TC.socket230VAC.terminals.N
49      4: ->
50        comment: "N out"
51    ) "circuitBreaker"
52
53
54  # Power distribution =====================================================
55
56  cover_elec.TC.ADD cont.contains CONTAINER(
57    items:
58      [
59        TERMINAL(
60          symbol: "PE"
61          comment: "Protective Earth"
62          isConnectedTo: cover_elec.TC.socket230VAC.terminals.PE) "PE"
63        TERMINAL(
64          symbol: "+24V"
65          comment: "+24VDC") "DC"
66        TERMINAL(
67          symbol: "GND"
68          comment: "GND") "GND"
69      ]
70    ) "terminals"
71
72
73  # DC supply ==============================================================
74
75  cover_elec.TC.ADD cont.contains POWER_SUPPLY_INSTANCE(
76    type: phoenix.trio_ps_1AC_24VDC_10
```

```
    symbol: "TC:PS24"
    comment: "24V power supply to power the I/O modules"
    terminals:
      PE: ->
        symbol: "PE"
        comment: "From PE terminals"
        isConnectedTo: cover_elec.TC.terminals.PE
      L: ->
        symbol: "L"
        comment: "From circuit breaker"
        isConnectedTo: cover_elec.TC.circuitBreaker.terminals[2]
      N: ->
        symbol: "N"
        comment: "From circuit breaker"
        isConnectedTo: cover_elec.TC.circuitBreaker.terminals[4]
      plus: ->
        symbol: "+"
        comment: "To +24V terminals"
        isConnectedTo: cover_elec.TC.terminals.DC
      minus: ->
        symbol: "-"
        comment: "To GND terminals"
        isConnectedTo: cover_elec.TC.terminals.GND
    ) "power"
```

Listing A.10: Electrical engineering model example (continued).

The telescope cover consists of 8 very similar panels, connected to the cabinet via 8 very similar connectors and 8 very similar cables. The variability between these 8 connector instances can thus be described by a parametric function, which we call 8 times to instantiate the connectors: see listing A.11.

```
# Connectors ============================================================

createConnector = (connectorName, petalName) ->
  CONNECTOR_INSTANCE(
    type: itt.Dsub15FS
    symbol: "TC:#{connectorName}"
    comment: "Connector to #{petalName}"
    terminals:
      1 : ->
        symbol: "TC:#{connectorName}:GND HM"
        comment: "#{petalName} GND of holding magnet"
        isConnectedTo: cover_elec.TC.terminals.GND
      2 : ->
        symbol: "TC:#{connectorName}:GND MOT"
        comment: "#{petalName} GND of motor"
        isConnectedTo: cover_elec.TC.terminals.GND
      3 : ->
        symbol: "TC:#{connectorName}:MMON"
        comment: "#{petalName} motor monitor"
      4 : ->
        symbol: "TC:#{connectorName}:MDIR"
        comment: "#{petalName} motor direction"
      5 : ->
        symbol: "TC:#{connectorName}:GND ENC"
        comment: "#{petalName} GND of encoder"
        isConnectedTo: cover_elec.TC.terminals.GND

        # other terminals not shown in this listing

    ) connectorName


cover_elec.TC.ADD cont.contains CONTAINER(
```

```
    items:
      [
        CONNECTOR_INSTANCE(
          type: harting.RJ45F
          symbol: "TC:ECAT"
          comment: "EtherCAT input, from junction"
        ) "ECAT"
        createConnector("T1", "Top 1")
        createConnector("T2", "Top 2")
        createConnector("T3", "Top 3")
        createConnector("T4", "Top 4")
        createConnector("B1", "Bottom 1")
        createConnector("B2", "Bottom 2")
        createConnector("B3", "Bottom 3")
        createConnector("B4", "Bottom 4")
      ]
    ) "connectors"
```

Listing A.11: Electrical engineering model example (continued).

In a similar way, we can define parametric functions to describe similar I/O modules, or we can describe the I/O modules directly. Listing A.12 shows a small part of the I/O module descriptions.

```
# I/O modules ============================================================

createSSIModule = (connector1, connector2, panel1, panel2) ->
  IO_MODULE_INSTANCE(
    comment   : "SSI module for #{panel1} and #{panel2} encoders"
    type      : beckhoff.EL5002
    terminals :
      1: ->
        symbol: "TC:#{connector1}:SSID+"
        comment: "#{panel1} SSI encoder Data +"
        isConnectedTo: cover_elec.TC.connectors[connector1].terminals[6]
      2: ->
        symbol: "TC:#{connector1}:SSIC+"
        comment: "#{panel1} SSI encoder Clock +"
        isConnectedTo: cover_elec.TC.connectors[connector1].terminals[7]
      3: ->
        symbol: "TC:#{connector2}:SSID+"
        comment: "#{panel2} SSI encoder Data +"
        isConnectedTo: cover_elec.TC.connectors[connector2].terminals[6]

      # remaining terminals not shown in this listing

  )


# other functions not shown in this listing

cover_elec.TC.ADD cont.contains CONTAINER(
  items:

    # slot0, slot1, slot2 not shown in this listing

    slot3: ->
      IO_MODULE_INSTANCE(
        type    : beckhoff.EL1088
        comment   : "Digital input terminal to read the status of the SSI
            encoders of all 8 cover panels"
        satisfies : cover_sys.panelDesign.requirements.absFeedbackStatus
        terminals :
          1: ->
            symbol: "TC:T1:SSISTS"
```

```
363               comment: "Top 1 SSI status"
364               isConnectedTo: cover_elec.TC.connectors.T1.terminals[13]
365             2: ->
366               symbol: "TC:T2:SSISTS"
367               comment: "Top 2 SSI status"
368               isConnectedTo: cover_elec.TC.connectors.T2.terminals[13]
369             3: ->
370               symbol: "TC:T3:SSISTS"
371               comment: "Top 3 SSI status"
372               isConnectedTo: cover_elec.TC.connectors.T3.terminals[13]
373             4: ->
374               symbol: "TC:T4:SSISTS"
375               comment: "Top 4 SSI status"
376               isConnectedTo: cover_elec.TC.connectors.T4.terminals[13]

              # remaining terminals not shown in this listing

393         )
394       slot4: -> createSSIModule('T1', 'T2', 'Top 1', 'Top 2')
395       slot5: -> createSSIModule('T3', 'T4', 'Top 3', 'Top 4')
396       slot6: -> createSSIModule('B1', 'B2', 'Bottom 1', 'Bottom 2')
397       slot7: -> createSSIModule('B3', 'B4', 'Bottom 3', 'Bottom 4')

      # remaining slots not shown in this listing

393   ) "io"
```

Listing A.12: Electrical engineering model example (continued).

Finally, we can model the "field" as a configuration, consisting of a cable assembly (a cable with a connector at each end), a socket to plug the cable into, and three devices (an encoder, a motor and a magnet) wired to this socket. In listing A.13 we first define the cable assembly as a custom type, manufactured by the Institute for Astronomy. For each petal, we can then instantiate and connect the cable assembly, socket, motor, encoder, and magnet types.

```
395 ################################################################################
396 # Field
397 ################################################################################
398
399 cover_elec.TC.ADD cont.contains elec.CONFIGURATION(
400   label: "field"
401   comment: "Contains everything outside the cabinet"
402   ) "field"
403
404 # cable assembly (=connector+cable+connector) common to all panels
405 cover_elec.ADD CABLE_ASSEMBLY_TYPE(
406   comment : "Cable assembly between cabinet and field"
407   id        : "CoverCableAssembly"
408   manufacturer  : ivs.organization
409   connectors:
410     cabinetSide: ->
411       type: itt.Dsub15MP
412       comment: "Plug on the cabinet side"
413     petalSide: ->
414       type: phoenix.MCVU_16_plug
415       comment: "Plug on the petal side"
416   cables:
417     cable     : ->
418       type: various.Cable15x034S
419       comment: "Cable between both plugs"
420       wires:
421         white: ->
422           from: $.connectors.cabinetSide.terminals[1]
```

```
            to   : $.connectors.petalSide.terminals[1]
          black: ->
            from: $.connectors.cabinetSide.terminals[9]
            to   : $.connectors.petalSide.terminals[2]
          brown: ->
            from: $.connectors.cabinetSide.terminals[2]
            to   : $.connectors.petalSide.terminals[3]

            # remaining wires not shown in this listing

    ) "CableAssembly"


  for panel in ["T1", "T2", "T3", "T4", "B1", "B2", "B3", "B4"]

    # create a configuration for the panel
    cover_elec.TC.field.ADD cont.contains elec.CONFIGURATION() "#{panel}"

    # create the cable assembly
    cover_elec.TC.field[panel].ADD cont.contains CABLE_ASSEMBLY_INSTANCE(
      comment  :  "Cable assembly of panel #{panel}"
      type     : cover_elec.CableAssembly
      joined:
        cabinetSide: -> cover_elec.TC.connectors[panel]
    ) "cableAssembly"

    # create the socket on the petal-side
    cover_elec.TC.field[panel].ADD cont.contains CONNECTOR_INSTANCE(
      comment     : "Field socket of #{panel}"
      type      : phoenix.MCVU_16_socket
      joinedWith:
        cover_elec.TC.field[panel].cableAssembly.connectors.petalSide
    ) "socket"

    # create the SSI encoder and wire it to the socket
    cover_elec.TC.field[panel].ADD cont.contains SENSOR_INSTANCE(
      comment     : "External encoder of panel #{panel}"
      realizes    : cover_sys.panelDesign.parts.encoder
      type      : kuebler.F3673_1421_G412
      cables:
        cable: ->
          wires:
            white  : -> to: cover_elec.TC.field[panel].socket.terminals[8]
            red    : -> to: cover_elec.TC.field[panel].socket.terminals[8]
            blue   : -> to: cover_elec.TC.field[panel].socket.terminals[8]
            brown  : -> to: cover_elec.TC.field[panel].socket.terminals[9]
            violet : -> to: cover_elec.TC.field[panel].socket.terminals[10]
            gray   : -> to: cover_elec.TC.field[panel].socket.terminals[11]
            pink   : -> to: cover_elec.TC.field[panel].socket.terminals[12]
            green  : -> to: cover_elec.TC.field[panel].socket.terminals[13]
            yellow : -> to: cover_elec.TC.field[panel].socket.terminals[14]
            screen : -> to: cover_elec.TC.field[panel].socket.terminals[16]
    ) "encoder"

    # create the motor and wire it to the socket
    cover_elec.TC.field[panel].ADD cont.contains elec.MOTOR_INSTANCE(
      comment   : "Motor of panel #{panel}",
      realizes  : cover_sys.panelDesign.parts.motor
      type      : maxon.motor_370418
      wires:
        black: -> to: cover_elec.TC.field[panel].socket.terminals[3]
        red  : -> to: cover_elec.TC.field[panel].socket.terminals[4]
        green: -> to: cover_elec.TC.field[panel].socket.terminals[5]
        white: -> to: cover_elec.TC.field[panel].socket.terminals[6]
        gray : -> to: cover_elec.TC.field[panel].socket.terminals[7]
    ) "motor"

    # create the magnet and wire it to the socket
    cover_elec.TC.field[panel].ADD cont.contains elec.ACTUATOR_INSTANCE(
      comment  : "Magnet"
```

```
581       realizes : cover_sys.panelDesign.parts.magnet
582       type     : magnetschultz.G_MH_x025
583       terminals:
584         GND:-> isConnectedTo: cover_elec.TC.field[panel].socket.terminals[1]
585         VCC:-> isConnectedTo: cover_elec.TC.field[panel].socket.terminals[2]
586     ) "magnet"
```

Listing A.13: Electrical engineering model example (continued).

A.4 Software engineering

The final model about the telescope cover subsystem (`cover_soft`) represents the software of the subsystem. We start by defining a software library (`mtcs_cover`) which contains all type definitions, and which can be converted into source code. All type definitions are created via custom macros, whose names begin with `MTCS_MAKE_`. These macros add application-specific features to the models: for instance, the `MTCS_MAKE_LIB` macro creates a `soft:Library` individual, sets the target language to IEC 61131-3, and adds namespaces such as `Enums`, `Statuses`, `StateMachines`, `Configs`, etc. In the TwinCAT 3 programming environment, these namespaces will appear as directories.

```
1  ##############################################################################
2  #                                                                            #
3  # Model of the cover software.                                               #
4  #                                                                            #
5  ##############################################################################
6
7  require "ontoscript"
8
9  # require the dependencies
10 REQUIRE "models/mtcs/common/software.coffee"
11 REQUIRE "models/util/softwarefactories.coffee"
12
13 MODEL "http://www.mercator.iac.es/onto/models/mtcs/cover/software" :
14       "cover_soft"
15
16 cover_soft.IMPORT common_soft
17
18 ##############################################################################
19 # Define the containing PLC library
20 ##############################################################################
21
22 cover_soft.ADD MTCS_MAKE_LIB "mtcs_cover"
23
24 # make aliases (with scope of this file only)
25 COMMONLIB = common_soft.mtcs_common
26 THISLIB   = cover_soft.mtcs_cover
```

Listing A.14: Software engineering model example.

Several more `MTCS_MAKE_...` macros have been defined, to create enumerations, configurations, state machines, and so on. Since the complete models are too extensive to be displayed in this thesis, we only show one example of each. Listing A.15 shows the definition of an enumeration of the states of the opening

and closing procedures of a petal. The `MTCS_MAKE_ENUM` macro assigns the newly created enumeration to the `Enums` namespace, which was created earlier.

```
27  ##########################################################################
28  # CoverApertureProcedureStates
29  ##########################################################################
30
31  MTCS_MAKE_ENUM THISLIB, "CoverApertureProcedureStates",
32    comment: "The disjoint states of the opening and closing procedure"
33    items:
34      [ "IDLE",
35        "ABORTED",
36        "PREPARE_PROCESS",
37        "ENABLING_RELAYS",
38        "ENABLING_MOTORS",
39        "ENABLING_MAGNETS",
40        "DISABLING_RELAYS",
41        "DISABLING_MOTORS",
42        "DISABLING_MAGNETS",
43        "OPENING_TOP_PANELS",
44        "OPENING_BOTH_PANELS",
45        "CLOSING_BOTTOM_PANELS",
46        "CLOSING_BOTH_PANELS",
47        "ERROR",
48        "RESETTING",
49        "ABORTING"
50      ]
```

Listing A.15: Software engineering model example (continued).

Listing A.16 shows the creation of three software configurations, via the `MTCS_MAKE_CONFIG` macro. This macro produces an `iec61131:Struct` instance, and assigns the instance to the `Configs` namespace of the library. As can be seen at lines 414-415, the items of a configuration do not have to be fully defined during the definition of the configuration. Type (and other) information can always be added later: in this case at lines 266-267. This allows us to work in a top-down way: from the high-level definition of `CoverConfig` to the lower level definition of `CoverPanelConfig`.

```
408 ##########################################################################
409 # CoverConfig
410 ##########################################################################
411
412 MTCS_MAKE_CONFIG THISLIB, "CoverConfig",
413   items:
414     top     : { comment: "The config of the top panel set" }
415     bottom  : { comment: "The config of the bottom panel set" }
416     openingVelocity:
417       comment: "The opening velocity of the panels in degrees per second"
418       type:  t_double
419     closingVelocity:
420       comment: "The closing velocity of the panels in degrees per second"
421       type:  t_double
422     magnetRemanentTime:
423       comment: "How many seconds should be waited after disabling magnets"
424       type:  t_double
    # remaining items not shown in this listing

259 ##########################################################################
260 # CoverPanelSetConfig
```

```
261  ##############################################################################
262
263  MTCS_MAKE_CONFIG THISLIB, "CoverPanelSetConfig",
264    typeOf: [ THISLIB.CoverParts.top.config,
265              THISLIB.CoverParts.bottom.config,
266              THISLIB.CoverConfig.top,
267              THISLIB.CoverConfig.bottom ]
268    items:
269      p1        : { comment: "The config of the first panel of this set" }
270      p2        : { comment: "The config of the second panel of this set" }
271      p3        : { comment: "The config of the third panel of this set" }
272      p4        : { comment: "The config of the fourth panel of this set" }
273      name      : { type: t_string  , comment: "The name of the panel set" }
         # remaining items not shown in this listing

400  ##############################################################################
401  # CoverPanelConfig
402  ##############################################################################
403
404  MTCS_MAKE_CONFIG THISLIB, "CoverPanelConfig",
405    typeOf: [THISLIB.CoverPanelSetConfig[p] for p in ["p1","p2","p3","p4"]]
406    items:
407      closedPosition:
408        type: t_double
409        comment: "The closed position of the panel in degrees"
410      openPosition:
411        type: t_double
412        comment: "The open position of the panel in degrees"
413      openTolerance:
414        type: t_double
415        comment: "The tolerance for opening, in degrees"
         # remaining items not shown in this listing
```

Listing A.16: Software engineering model example (continued).

Finally, the `MTCS_MAKE_STATEMACHINE` macro creates two IEC 61131-3 function blocks, to represent a state machine. For instance, the macro call shown in listing A.17 produces the function blocks `SM_CoverPanel` and `CoverPanel`. The former holds all declared variables and implementation details that are modeled in Ontoscript, while the latter is an "empty" function block that extends (i.e. that is a subtype of) the former. Since only `SM_CoverPanel` is added to the `StateMachine` namespace of the `mtcs_cover` library, only this function block will eventually be converted into source code. The extending IEC 61131-3 function block called `CoverPanel` is not contained by the library, and hence need to be created manually, in the TwinCAT programming environment. Any "custom" (non-reusable) code can therefore be implemented within the `CoverPanel` function block, while the structure, the most important states, and the execution order of the software are determined by the Ontoscript model.

Below we explain the most important slots of the `MTCS_MAKE_STATEMACHINE` macro, via the example code of listing A.17:

- `typeOf`: produces `soft:isTypeOf` relations, to assert that the produced function block is the type of the given earlier defined variables.
- `references`: produces IEC 61131-3 input/output variables (similar to references in C++).
- `variables`: produces IEC 61131-3 input variables.

- `parts`: produces an IEC61131-3 output variable called `parts` of type `CoverPanelParts`. The latter is an automatically generated IEC 61131-3 structure that declares the "sub-statemachines" `axis` and `motorRelay`.
- `statuses`: produces an IEC61131-3 output variable called `statuses` of type `CoverPanelStatuses`. The latter is an automatically generated IEC 61131-3 structure that declares the "status" instances `busyStatus`, `healthStatus`, etc.
- `processes`: produces an IEC61131-3 output variable called `processes` of type `CoverPanelProcesses`. The latter is an automatically generated IEC 61131-3 structure that declares the processes `startOpening` and `startClosing`, and the corresponding IEC 61131-3 methods.
- `calls`: adds the specified function block calls to the implementation of the `SM_CoverPanel` function block, in a well defined order. Any parts that are not explicitly called, will be called automatically without arguments.

```
436  ##############################################################################
437  # CoverPanel
438  ##############################################################################
439
440  MTCS_MAKE_STATEMACHINE THISLIB, "CoverPanel",
441    typeOf: [ THISLIB.CoverPanelSetParts.p1,
442              THISLIB.CoverPanelSetParts.p2,
443              THISLIB.CoverPanelSetParts.p3,
444              THISLIB.CoverPanelSetParts.p4,
445              THISLIB.CoverPanelSet.parts.p1,
446              THISLIB.CoverPanelSet.parts.p2,
447              THISLIB.CoverPanelSet.parts.p3,
448              THISLIB.CoverPanelSet.parts.p4 ]
449    references:
450      initializationStatus:
451          type: COMMONLIB.InitializationStatus
452          comment: "INITIALIZED or INITIALIZING or ..."
453      operatorStatus:
454          type: COMMONLIB.OperatorStatus
455          comment: "TECH or OBSERVER or ..."
456      operatingStatus:
457          type: COMMONLIB.OperatingStatus
458          comment: "MANUAL or AUTO or NONE"
459      config:
460          type: THISLIB.CoverPanelConfig
461          comment: "Configuration of the panel"
462      coverConfig:
463          type: THISLIB.CoverConfig
464          comment: "Configuration of the cover"
465          expand: false
466    variables:
467      encoderErrorSignal:
468          type: t_bool
469          comment: 'Externally read error signal'
470          address: "I*"
471    parts:
472      axis:
473          type: COMMONLIB.AngularAxis
474          comment: "NC Axis"
475      motorRelay:
476          type: COMMONLIB.SimpleRelay
477          comment: "Relay for the motor"
478    statuses:
```

```
    busyStatus:
        type: COMMONLIB.BusyStatus
        comment: "Is the panel in a busy state?"
    apertureStatus:
        type: COMMONLIB.ApertureStatus
        comment: "Is the panel open or closed?"
    healthStatus:
        type: COMMONLIB.HealthStatus
        comment: "Is the panel in a healthy state?"
    openingStatus:
        type: COMMONLIB.OpeningStatus
        comment: "Is the panel opening or closing or standing still?"
  processes:
    startOpening:
        type: COMMONLIB.Process
        comment: "Start opening the panel"
    startClosing:
        type: COMMONLIB.Process
        comment: "Start closing the panel"
  calls:
    axis:
      isEnabled: -> AND(self.operatorStatus.tech,
                        self.operatingStatus.manual,
                        self.initializationStatus.initialized)
      standstillTolerance : -> self.config.standstillTolerance
    motorRelay:
      isEnabled: -> self.parts.axis.isEnabled # same as for axis
    busyStatus:
      isBusy: -> OR( self.parts.axis.statuses.busyStatus.busy,
                     self.parts.motorRelay.statuses.busyStatus.busy )
    healthStatus:
      isGood: -> AND(self.parts.axis.statuses.healthStatus.isGood,
                     NOT(self.encoderErrorSignal))
      hasWarning: -> self.parts.axis.statuses.healthStatus.hasWarning
    apertureStatus:
      isOpen    : -> LT(ABS(SUB(self.config.openPosition,
                                self.parts.axis.actPos.degrees.value)),
                        self.config.openTolerance)
      isClosed  : -> LT(ABS(SUB(self.config.closedPosition,
                                self.parts.axis.actPos.degrees.value)),
                        self.config.closedTolerance)
    openingStatus:
      isOpening       : -> self.parts.axis.statuses.motionStatus.backward
      isClosing       : -> self.parts.axis.statuses.motionStatus.forward
    startOpening:
      isEnabled       : -> AND(self.operatorStatus.tech,
                               self.operatingStatus.manual,
                               self.initializationStatus.initialized)
    startClosing:
      isEnabled       : -> AND(self.operatorStatus.tech,
                               self.operatingStatus.manual,
                               self.initializationStatus.initialized)
```

Listing A.17: Software engineering model example (continued).

Appendix B

Appendix: OntoManager screen captures

In this appendix, a number of screen captures of OntoManager will be displayed, to illustrate the capabilities of our framework from an end-user perspective.

B.1 Login page

OntoManager is a multi-user application, requiring its users to login first.

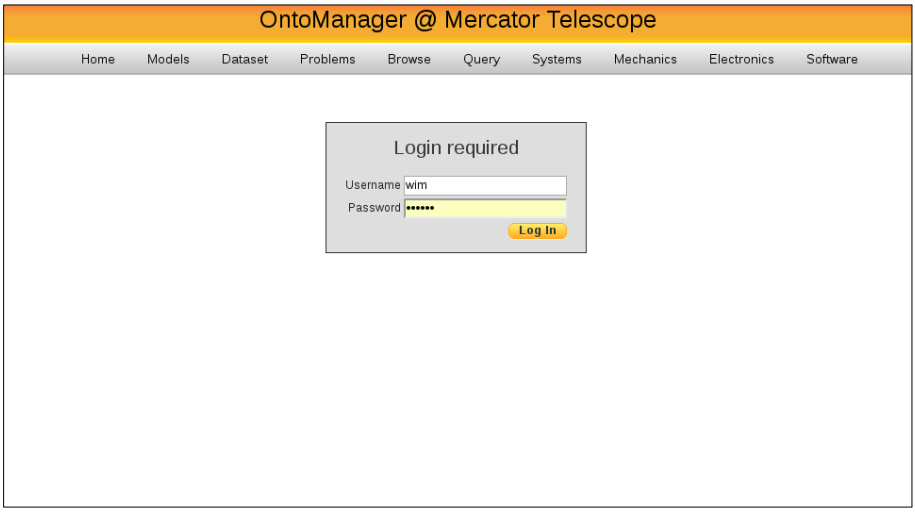


Figure B.1: OntoManager login page.

B.2 Home page

Once logged in, users are directed to their home page.

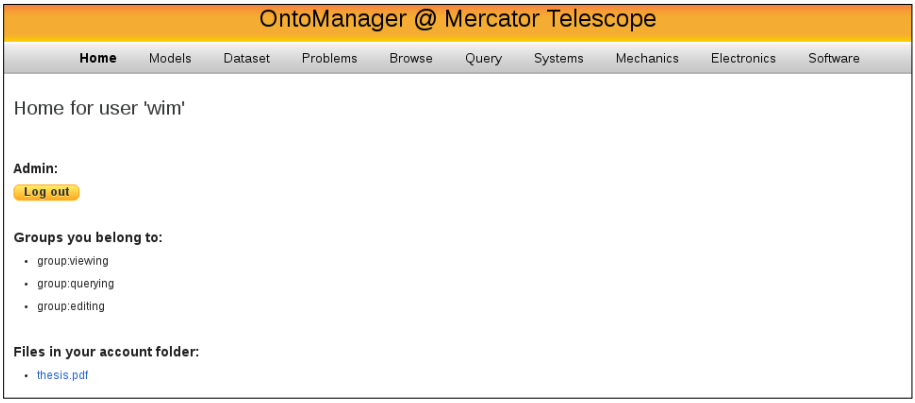


Figure B.2: OntoManager home page.

B.3 Models tab

Via the *Models* tab, users can inspect the existing Ontoscript models. Models are currently read-only, but future versions of OntoManager should offer write access, and some basic version control and file manipulation functionality.

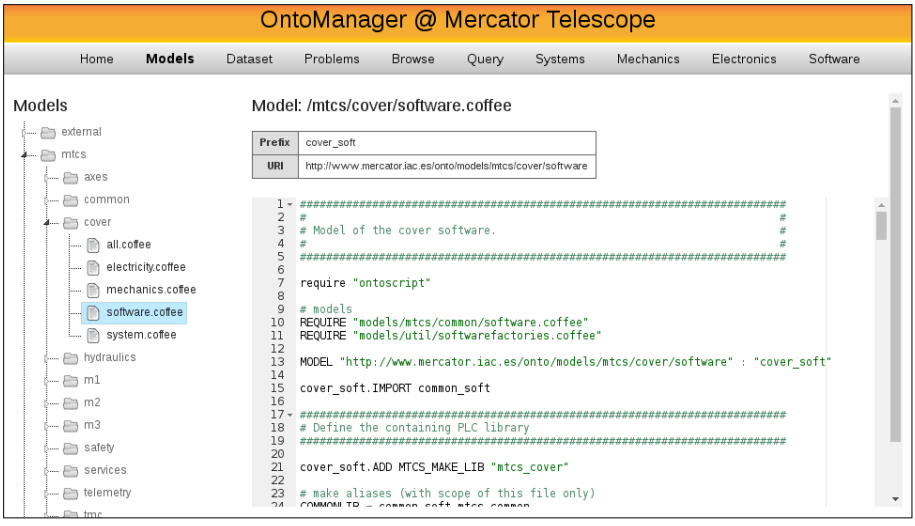


Figure B.3: OntoManager Models tab.

B.4 Dataset tab

Using the *Dataset* page, users can execute Ontoscript models, command the built-in reasoner, load data into memory, generate source code files, and load the cache memory from disk or save it persistently on disk.

Figure B.4: OntoManager Dataset tab.

B.5 Problems tab

Using the *Problems* tab, users can inspect constraint violations (after inferences have been produced and loaded into memory).

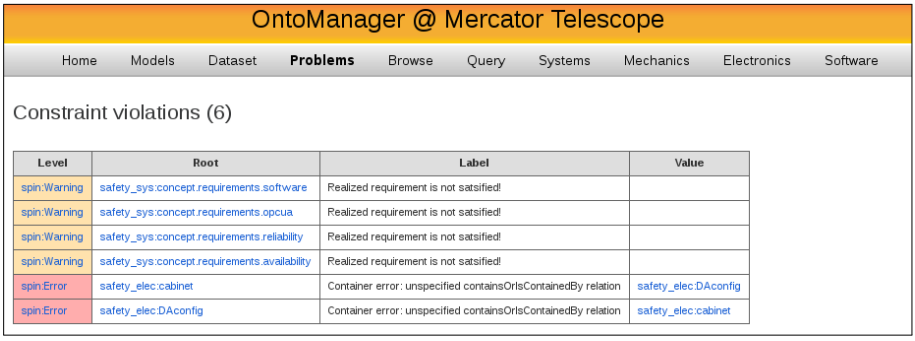


Figure B.5: OntoManager Problems tab.

B.6 Browse tab

Using the *Browse* tab, users can browse (i.e. list all facts of) any resource.

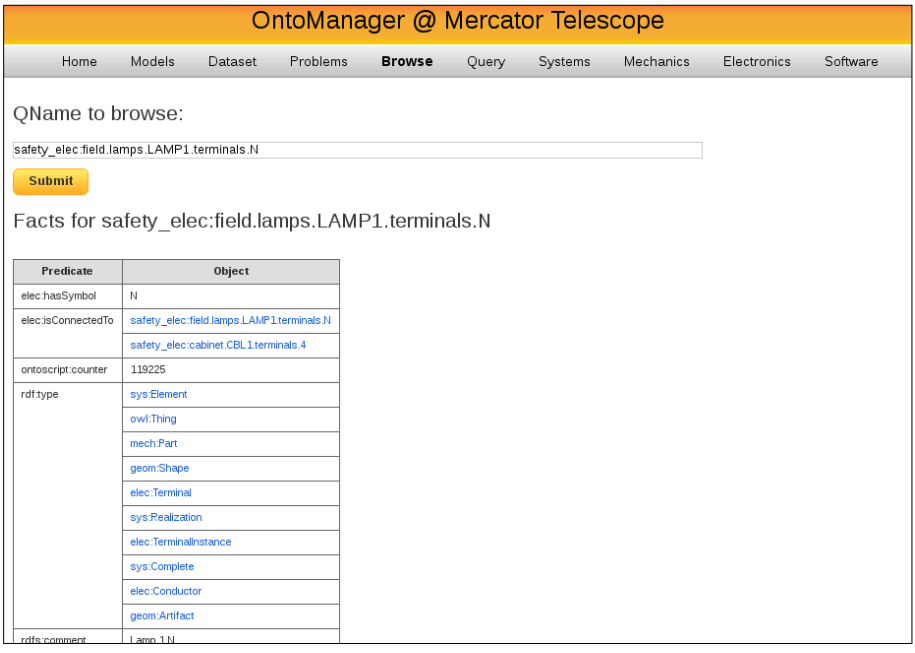


Figure B.6: OntoManager Browse tab.

B.7 Query tab

Using the *Query* tab, users can freely execute queries on the knowledge base. In the example below (figure B.7), a query has been executed to list all I/O modules that were used by the TCS, and how many times they were used.

OntoManager @ Mercator Telescope

Home Models Dataset Problems Browse **Query** Systems Mechanics Electronics Software

Query

```
SELECT ?company ?companyName ?m ?id ?comment (COUNT(?instance) AS ?count)
WHERE {
    ?m        rdf:type/rdfs:subClassOf*   elec:IoModuleType .
    ?m        man:hasId                   ?id .
    ?m        man:isManufacturedBy        ?company .
    ?company  org:hasLongName             ?companyName  .
    ?instance sys:realizes                ?m .
    OPTIONAL { ?m rdfs:comment ?comment }
}
GROUP BY ?id
ORDER BY ?companyName ASC(?id)
```

Submit

company	companyName	m	id	comment	count
beckhoff:company	Beckhoff Automation	beckhoff:EK1101	EK1101	EtherCAT Coupler with ID switch	9
beckhoff:company	Beckhoff Automation	beckhoff:EL1008	EL1008	8-channel digital input terminal 24V DC	4
beckhoff:company	Beckhoff Automation	beckhoff:EL1088	EL1088	8-channel digital input terminal 24V DC, negative switching	2
beckhoff:company	Beckhoff Automation	beckhoff:EL1904	EL1904	4-channel digital input terminal, TwinSAFE 24V DC	8
beckhoff:company	Beckhoff Automation	beckhoff:EL2008	EL2008	8-channel digital output terminal 24V DC	3
beckhoff:company	Beckhoff Automation	beckhoff:EL2024	EL2024	4-channel digital output terminals 24 V DC, 2 A	1
beckhoff:company	Beckhoff Automation	beckhoff:EL2124	EL2124	4-channel digital output terminals 5 V DC	2
beckhoff:company	Beckhoff Automation	beckhoff:EL2622	EL2622	2-channel relay	8
beckhoff:company	Beckhoff Automation	beckhoff:EL2904	EL2904	4-channel digital output terminal, TwinSAFE, 24V DC	1
beckhoff:company	Beckhoff Automation	beckhoff:EL3024	EL3024	4-channel analog input terminals 4...20mA, differential inputs, 12 bit	3
beckhoff:company	Beckhoff Automation	beckhoff:EL3102	EL3102	2-channel analog input terminals -10...+10 V, differential input, 16 bit	2
beckhoff:company	Beckhoff Automation	beckhoff:EL3152	EL3152	2-channel analog input terminal 4..20 mA, single-ended, 16 bit	1
beckhoff:company	Beckhoff Automation	beckhoff:EL3164	EL3164	4-channel analog input terminal 0...10 V, single-ended, 16 bit	1
beckhoff:company	Beckhoff Automation	beckhoff:EL3202_0010	EL3202-0010	2-channel input terminals PT100 (RTD) for 4-wire connection, high-precision	8
beckhoff:company	Beckhoff Automation	beckhoff:EL3351	EL3351	1-channel resistor bridge terminal (strain gauge)	3
beckhoff:company	Beckhoff Automation	beckhoff:EL3681	EL3681	Digital multimeter	1
beckhoff:company	Beckhoff Automation	beckhoff:EL4008	EL4008	8-channel analog output terminal 0...10V, 12 bit	1
beckhoff:company	Beckhoff Automation	beckhoff:EL4022	EL4022	2-channel analog output terminal 4...20 mA, 12 bit	1
beckhoff:company	Beckhoff Automation	beckhoff:EL4132	EL4132	2-channel analog output terminal -10...+10V, 16 bit	1
beckhoff:company	Beckhoff Automation	beckhoff:EL5001	EL5001	1-channel SSI encoder	2
beckhoff:company	Beckhoff Automation	beckhoff:EL5002	EL5002	2-channel SSI encoder	5
beckhoff:company	Beckhoff Automation	beckhoff:EL5101	EL5101	1-channel incremental encoder	8
beckhoff:company	Beckhoff Automation	beckhoff:EL6001	EL6001	RS-232 serial communication	1
beckhoff:company	Beckhoff Automation	beckhoff:EL6688	EL6688	IEEE 1588 external synchronisation interface	1
beckhoff:company	Beckhoff Automation	beckhoff:EL6751	EL6751	CANopen master/slave controller	1
beckhoff:company	Beckhoff Automation	beckhoff:EL6900	EL6900	TwinSAFE Logic	1
beckhoff:company	Beckhoff Automation	beckhoff:EL9070	EL9070	Shield terminal	3
beckhoff:company	Beckhoff Automation	beckhoff:EL9186	EL9186	Potential distribution terminal, 8 x 24V	2
beckhoff:company	Beckhoff Automation	beckhoff:EL9187	EL9187	Potential distribution terminal, 8 x 0V	3
beckhoff:company	Beckhoff Automation	beckhoff:EL9410	EL9410	Power supply terminals for E-bus (with diagnostics)	2
beckhoff:company	Beckhoff Automation	beckhoff:EL9505	EL9505	Power supply terminals 5 V	1

Figure B.7: OntoManager Query tab.

B.8 Systems tab

The *Systems* tab provides predefined views of the systems engineering models (with views for `dev:Concept`, `dev:Design`, `dev:Requirement`, ...).

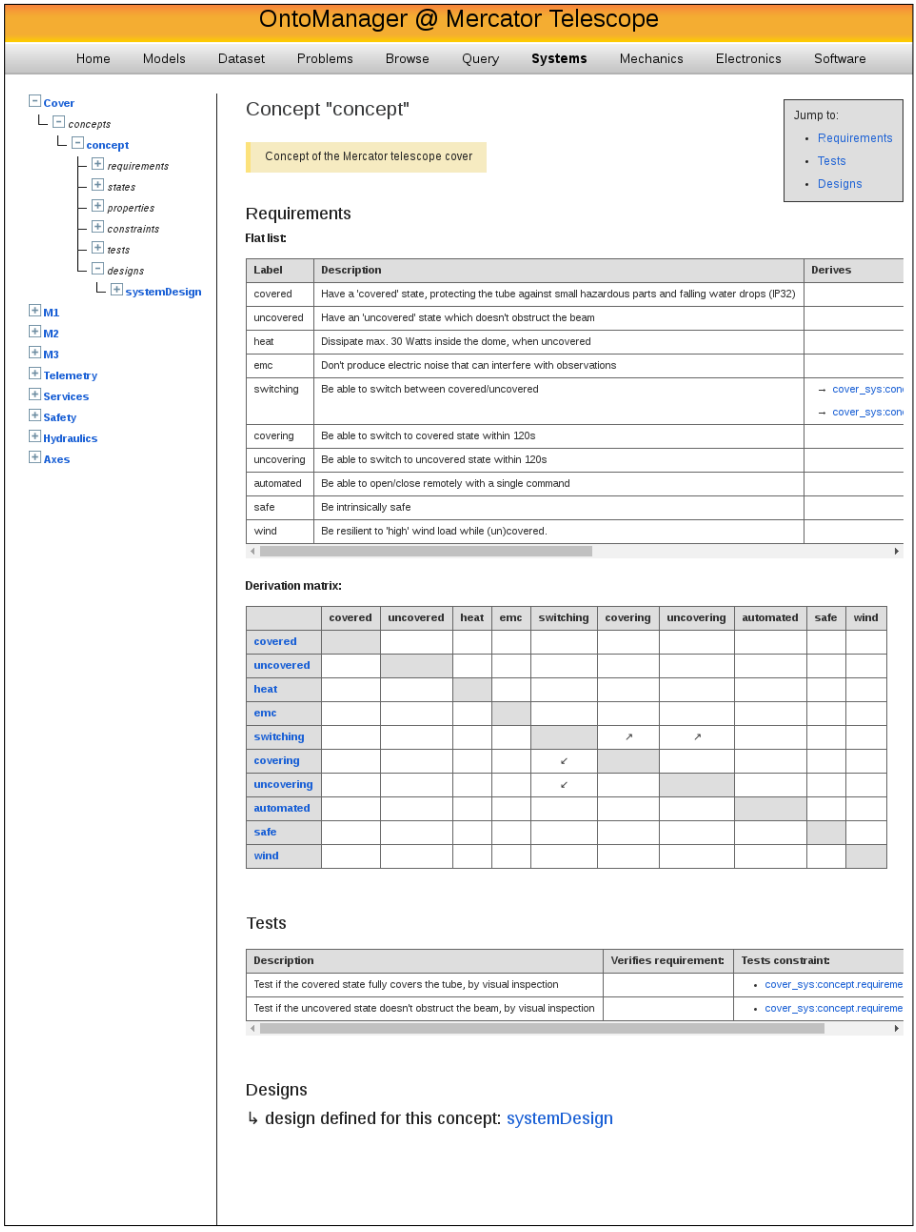


Figure B.8: OntoManager Systems tab.

B.9 Electronics tab

The *Electronics* tab offers views on the electrical systems of the telescope. Figure B.9 shows the view of an `elec:Configuration`, and figure B.10 of an `elec:IoModule` instance.

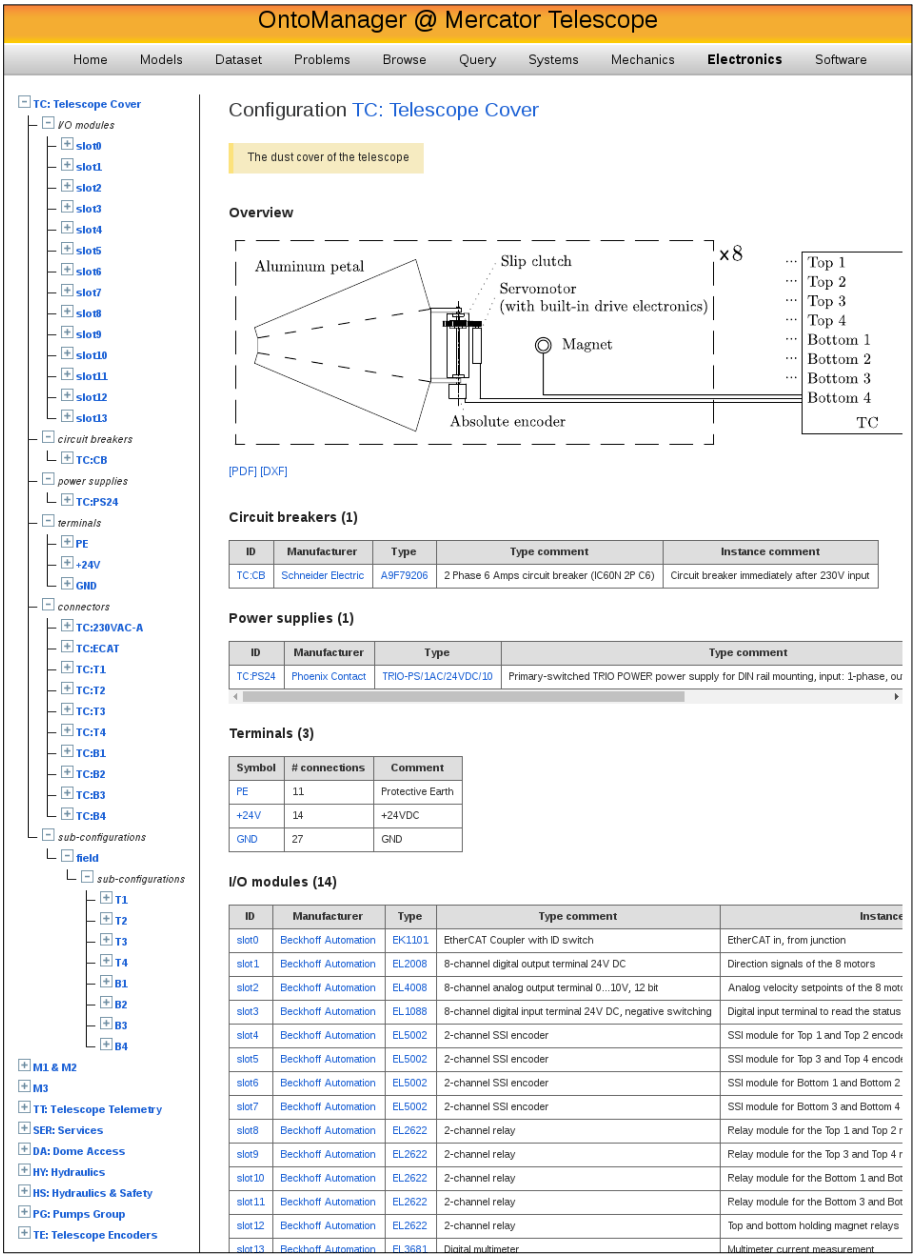


Figure B.9: OntoManager Electronics tab: Configuration view.

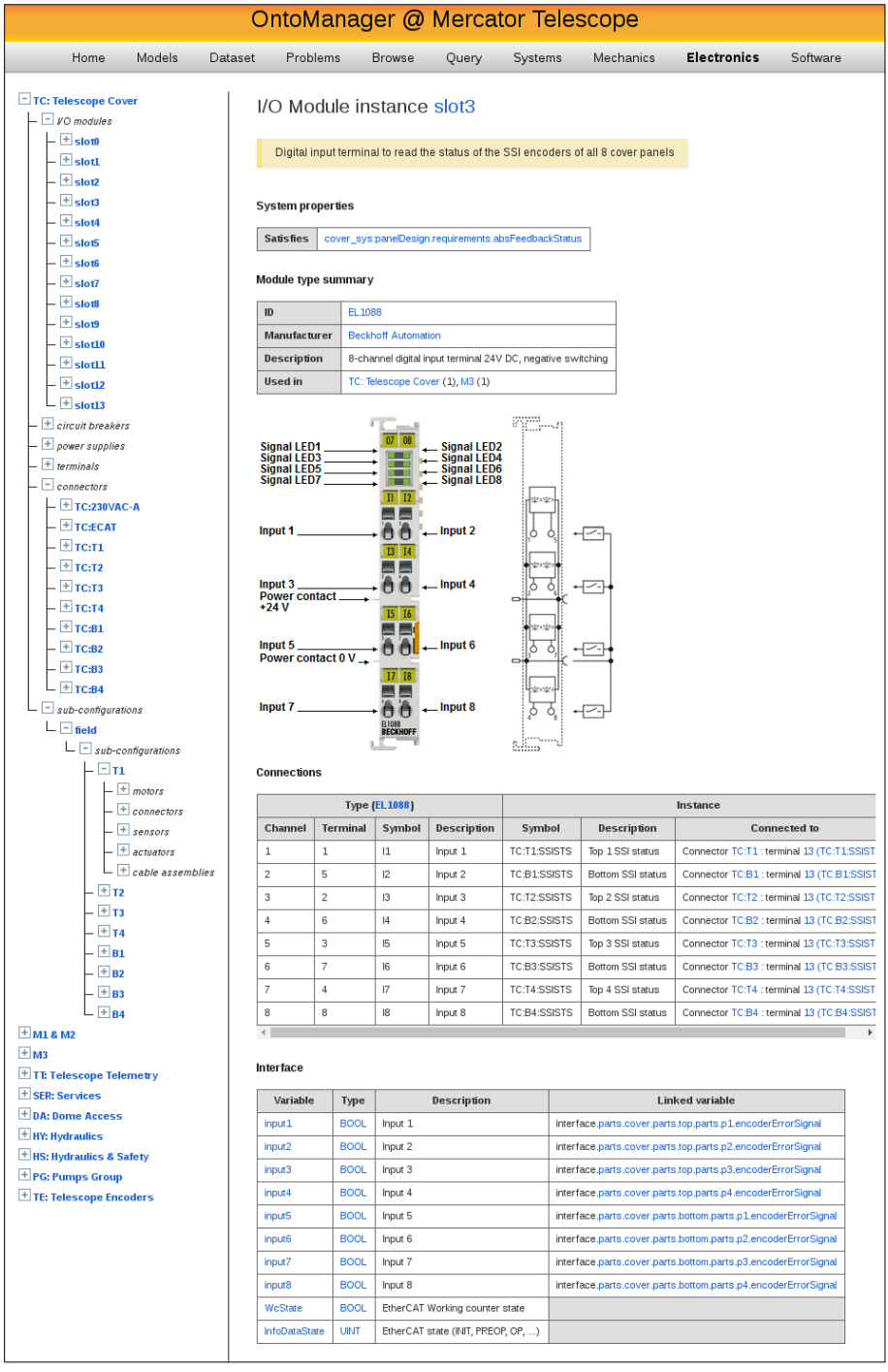


Figure B.10: OntoManager Electronics tab: I/O module view.

B.10 Software tab

The *Software* tab offers views on the software of the telescope. Figure B.11 shows the view of a `soft:Library`, with the possibility to download source code (either in PLCopen XML format to import in TwinCAT 3, or as Python code to import in MOCS). Figure B.12 shows the HMTL documentation of an IEC 61131-3 function block. One can see that every individual variable of the *Implementation* section can be clicked in the web-browser, which demonstrates that every expression of the implementation is fully modeled – and not just "plain text" produced by the code generator, as it is often the case in current practices.

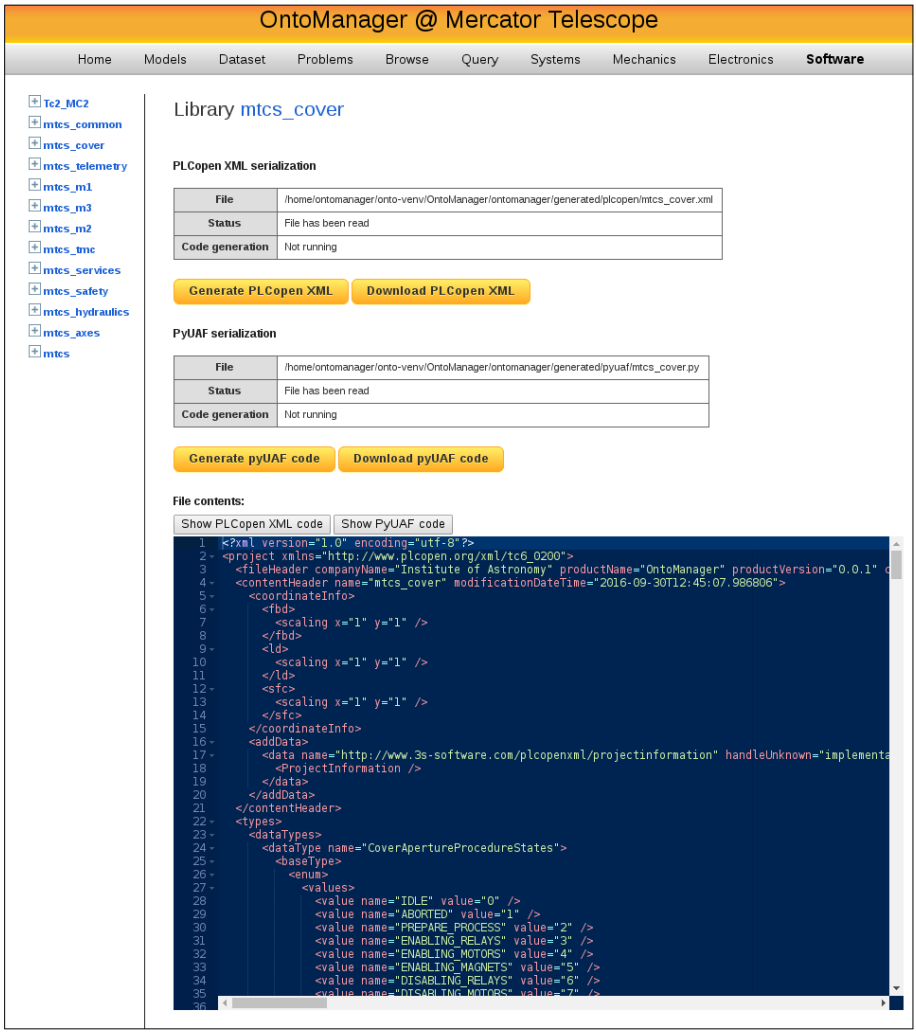


Figure B.11: OntoManager Software tab: Library view.

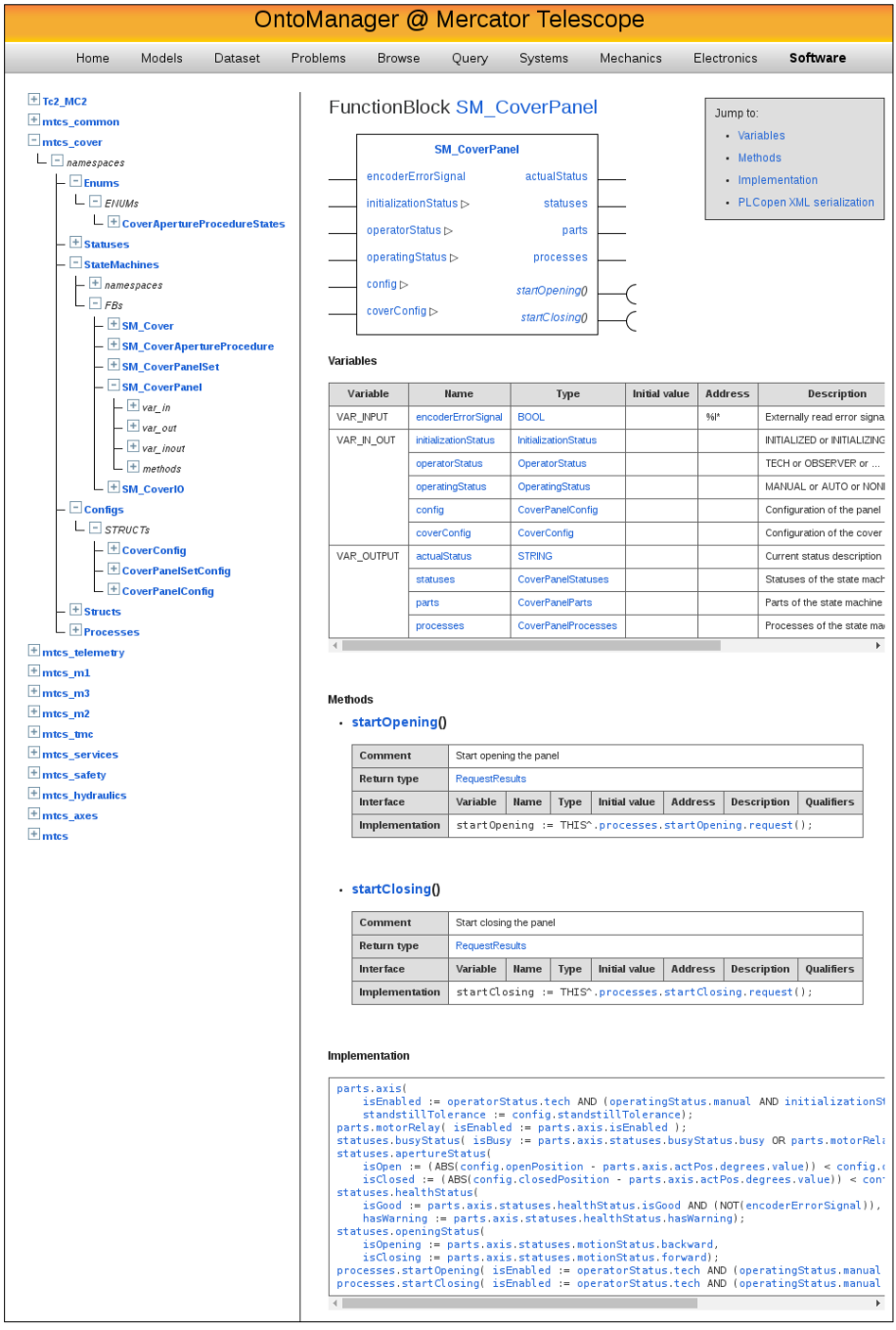


Figure B.12: OntoManager Software tab: Function block view.

References

[1] Ackoff, R. L. Towards a system of systems concepts. *Management science vol. 17*, no. 11 (1971), pp. 661–671.

[2] Ackoff, R. L. From data to wisdom. *Journal of applied systems analysis vol. 16*, no. 1 (1989), pp. 3–9.

[3] Bely, P. Y. *The design and construction of large optical telescopes*. Astronomy and astrophysics library. Springer-Verlag New York, Inc., 2003.

[4] Berners-Lee, T., Hendler, J., and Lassila, O. The semantic web. *Scientific American vol. 284*, no. 5 (May 2001), pp. 34–43.

[5] Bézivin, J. On the unification power of models. *Software & Systems Modeling vol. 4*, no. 2 (2005), pp. 171–188.

[6] Bézivin, J., and Gerbé, O. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering, 2001 (ASE 2001)* (Nov 2001), IEEE, pp. 273–280.

[7] Blockmans, B., Bielen, P., and Ceulemans, G. Finite element analysis of the primary mirror of the Mercator telescope. Master's thesis, Group T – Leuven Engineering College, 2010.

[8] Booch, G. *Object-oriented analysis and design with applications*, third ed. Addison Wesley Longman, 2007.

[9] Borst, W. N. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, Enschede, September 1997.

[10] Brooke, J., et al. SUS – A quick and dirty usability scale. *Usability Evaluation in Industry vol. 189*, no. 194 (1996), pp. 4–7.

[11] Broy, M., and Cengarle, M. V. UML formal semantics: lessons learned. *Software and Systems Modeling vol. 10*, no. 4 (2011), pp. 441–446.

[12] Cabrera, A. A., Foeken, M., Tekin, O., Woestenenk, K., Erden, M., Schutter, B. D., van Tooren, M., Babuška, R., van Houten,

F., and Tomiyama, T. Towards automation of control software: A review of challenges in mechatronic design. *Mechatronics vol. 20*, no. 8 (2010), pp. 876 – 886.

[13] Cena, G., Bertolotti, I. C., Scanzio, S., Valenzano, A., and Zunino, C. Evaluation of EtherCAT Distributed Clock performance. *IEEE Transactions on Industrial Informatics vol. 8*, no. 1 (2012), pp. 20–29.

[14] Chen, H., Wu, Z., and Cudre-Mauroux, P. Semantic Web Meets Computational Intelligence: State of the Art and Perspectives [Review Article]. *IEEE Computational Intelligence Magazine vol. 7*, no. 2 (2012), pp. 67–74.

[15] Chiozzi, G., Jeram, B., Sommer, H., Caproni, A., Plesko, M., Sekoranja, M., Zagar, K., Fugate, D. W., Di Marcantonio, P., and Cirami, R. The ALMA common software: a developer-friendly CORBA-based framework. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2004), Conference 5496, Advanced Software, Control, and Communication Systems for Astronomy, International Society for Optics and Photonics (SPIE), pp. 205–218.

[16] Chiozzi, G., Wallander, A., Gillies, K., Goodrich, B., Wampler, S., Johnson, J., McCann, K., Schumacher, G., and Silva, D. Trends in software for large astronomy projects. In *Proceedings of the 11th International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)* (2007).

[17] Claver, C. F., Selvy, B. M., Angeli, G., Delgado, F., Dubois-Felsmann, G., Hascall, P., Lotz, P., Marshall, S., Schumacher, G., and Sebag, J. Systems engineering in the Large Synoptic Survey Telescope project: an application of model based systems engineering. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2014), Conference 9150, Modeling, Systems Engineering, and Project Management for Astronomy VI, p. 91500M.

[18] Collins English Dictionary. `http://www.collinsdictionary.com`, 2016. [Online; accessed 22 September 2016].

[19] Dalesio, L. R., Hill, J. O., Kraimer, M., Lewis, S., Murray, D., Hunt, S., Watson, W., Clausen, M., and Dalesio, J. The Experimental Physics and Industrial Control System architecture: past, present, and future. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment vol. 352*, no. 1 (1994), pp. 179–184.

[20] Davignon, G., Blecha, A., Burki, G., Carrier, F., Groenewegen, M., Maire, C., Raskin, G., Van Winckel, H., and Weber, L. CCD camera and automatic data reduction pipeline for the Mercator telescope on La Palma. In *Proceedings of SPIE Astronomical Telescopes and*

Instrumentation (2004), Conference 5492, Ground-based Instrumentation for Astronomy, International Society for Optics and Photonics (SPIE), pp. 871–879.

[21] Dekens, F. G., Seo, B.-J., and Troy, M. System modeling of the thirty meter telescope alignment and phasing system. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2014), Conference 9150, Modeling, Systems Engineering, and Project Management for Astronomy VI, p. 91500Z.

[22] Derezińska, A., and Pilitowski, R. Interpretation of History Pseudostates in Orthogonal States of UML State Machines. In *Next Generation Information Technologies and Systems: 7th International Conference, NGITS 2009. Revised Selected Papers* (2009), Springer-Verlag, Berlin Heidelberg, pp. 26–37.

[23] Dorf, R. C., and Bishop, R. H. *Modern Control Systems*, 12th ed. Pearson Prentice Hall, 2011.

[24] El Kouhen, A., Gherbi, A., Dumoulin, C., and Khendek, F. On the Semantic Transparency of Visual Notations: Experiments with UML. In *Proceedings of the 17th International SDL Forum* (May 2015), Springer-Verlag, pp. 122–137.

[25] European Southern Observatory. *The E-ELT construction proposal*. 2011.

[26] Favre, J.-M. Megamodelling and etymology. In *Dagstuhl Seminar Proceedings* (2006), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[27] Fernández-López, M., Gómez-Pérez, A., and Juristo, N. Methontology: from ontological art towards ontological engineering. In *AAAI-97 Spring Symposium Series* (March 1997), American Asociation for Artificial Intelligence.

[28] Filgueira, J. M., Bec, M., Soto, J., Liu, N., and Peng, C. Y. GMT software and controls overview. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2012), Conference 8451, Software and Cyberinfrastructure for Astronomy II, International Society for Optics and Photonics (SPIE), p. 845111.

[29] Flouris, G., Huang, Z., Pan, J. Z., Plexousakis, D., and Wache, H. Inconsistencies, negations and changes in ontologies. In *Proceedings of the National Conference on Artificial Intelligence* (2006), vol. 21, p. 1295.

[30] Fowler, M. How standard is Standard UML? Originally published in Distributed Computing, Spring 1999, online `http://www.martinfowler.com/distributedComputing/standard.html`, 1999. [Online; accessed 20 September 2016].

[31] Gašević, D., Djurić, D., and Devedžić, V. *Model Driven Engineering and Ontology Development*, second ed. Springer-Verlag, Berlin Heidelberg, 2009.

[32] Genesereth, M. R., and Nilsson, N. J. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, 1987.

[33] Glinz, M. On Non-Functional Requirements. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE 2007)* (Oct 2007), IEEE, pp. 21–26.

[34] GMTO Corporation. GMT System level preliminary design review. Tech. rep., Giant Magellan Telescope Organization, 2013. Technical overview, Section 3.

[35] Gomaa, H. *Real-Time Software Design for Embedded Systems*. Cambridge University Press, 2016.

[36] Gotel, O. C., and Finkelstein, C. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering (RE 1994)* (1994), IEEE, pp. 94–101.

[37] Gruber, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition vol. 5*, no. 2 (1993), pp. 199–220.

[38] Gruber, T. R. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies vol. 43*, no. 5 (1995), pp. 907–928.

[39] Guarino, N., Oberle, D., and Staab, S. What is an ontology? *Handbook on Ontologies* (2009), pp. 1–17.

[40] Guzman, J., Chiozzi, G., Bridger, A., and Ibsend, J. The cost of developing and maintain the monitoring and control software of large ground-based telescopes. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2014), Conference 9152, Software and Cyberinfrastructure for Astronomy III, International Society for Optics and Photonics (SPIE), p. 91521P.

[41] Harel, D., and Rumpe, B. Meaningful modeling: what's the semantics of "semantics"? *Computer Journal vol. 37*, no. 10 (2004), pp. 64–72.

[42] Hebeler, J., Fisher, M., Blace, R., Perez-Lopez, A., and Dean, M. *Semantic Web Programming*. Wiley Publishing, 2009.

[43] Henderson-Sellers, B. UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and Systems Modeling vol. 4*, no. 1 (2005), pp. 4–13.

[44] Henderson-Sellers, B., and Barbier, F. Black and white diamonds. *UML '99 — The Unified Modeling Language: Beyond the Standard Second International Conference Fort Collins, CO, USA, October 28–30, 1999* (1999), pp. 550–565.

[45] Hodgson, R., Spivak, J., Ray, S., Hodges, J., and Keller, P. J. QUDT - Quantities, Units, Dimensions and Data Types Ontologies, September 18, 2016. `http://qudt.org`, 2016. [Online; accessed 26 October 2016].

[46] Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Web Semantics: Science, Services and Agents on the World Wide Web vol. 1*, no. 1 (2003), pp. 7–26.

[47] ICALEPCS. About ICALEPCS. `http://www.icalepcs.org/icalepcs.html`, 2016. [Online; accessed 27 August 2016].

[48] Jost, A. An introduction to IEEE 1588. Instrument Control System Seminar at European Southern Observatory (ESO), 20-24 October, 2014.

[49] Karban, R. MBSE meeting at the SPIE Astronomical Instrumentation conference, Edinburgh, June 2016. Minutes at `http://www.omgwiki.org/MBSE/doku.php?id=mbse:telescope_mbse_sig_meetings_edingburgh_june_2016`. [Online; accessed 13 September 2016].

[50] Karban, R., Andolfato, L., Bristow, P., Chiozzi, G., Esselborn, M., Schilling, M., Schmid, C., Sommer, H., and Zamparelli, M. Model based systems engineering for astronomical projects. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2014), Conference 9150, Modeling, Systems Engineering, and Project Management for Astronomy VI, International Society for Optics and Photonics (SPIE), p. 91500L.

[51] Karban, R., Dekens, F. G., Herzig, S., Elaasar, M., and Jankevičius, N. Creating system engineering products with executable models in a model-based engineering environment. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2016), Conference 9911, Modeling, Systems Engineering, and Project Management for Astronomy VI, International Society for Optics and Photonics (SPIE), p. 99110B.

[52] Karban, R., Kornweibel, N., Dvorak, D., Ingham, M., and Wagner, D. Towards a State Based Control Architecture for Large Telescopes: Laying a Foundation at the VLT. In *Proceedings of the 13th International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)* (2011).

[53] Karban, R., Zamparelli, M., Bauvir, B., and Chiozzi, G. Three years of MBSE for a large scientific programme: Report from the Trenches

of Telescope Modeling. *INCOSE International Symposium vol. 22*, no. 1 (2012), pp. 1544–1558.

[54] Knublauch, H., Hendler, J. A., and Idehen, K. SPIN – Overview and Motivation, W3C Member Submission 22 February 2011. `https://www.w3.org/Submission/2011/SUBM-spin-overview-20110222/`, 2012. [Online; accessed 26 September 2016].

[55] Kotis, K., and Vouros, G. A. Human-centered ontology engineering: The HCOME methodology. *Knowledge and Information Systems vol. 10*, no. 1 (2006), pp. 109–131.

[56] Kühne, T. Matters of (meta-) modeling. *Software & Systems Modeling vol. 5*, no. 4 (2006), pp. 369–385.

[57] Larkin, J. H., and Simon, H. A. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science vol. 11*, no. 1 (1987), pp. 65–100.

[58] Lee, E. A. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)* (May 2008), pp. 363–369.

[59] Lee, E. A. Disciplined heterogeneous modeling. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems* (2010), Springer, pp. 273–287.

[60] Lee, E. A., and Seshia, S. A. *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*, second ed. Online at `http://leeseshia.org`, 2015.

[61] MacCaw, A. *The Little Book on CoffeeScript*. O'Reilly Media, Inc., 2012.

[62] Maler, O., and Nickovic, D. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[63] Marchal, J. On the Concept of a System. *Philosophy of Science* (1975), pp. 448–468.

[64] Martin, F. *Domain-specific Languages*. Addison-Wesley Professional, Boston, MA, 2010, ch. 2, pp. 27–42.

[65] Martin, R. C. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2003.

[66] Meditskos, G., and Bassilliades, N. Rule-based OWL ontology reasoning systems: Implementations, strength, and weakness. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches* (2009), pp. 124–148.

[67] Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR) vol. 37*, no. 4 (2005), pp. 316–344.

[68] Merriam-Webster Online. `http://www.merriam-webster.com`, 2016. [Online; accessed 22 September 2016].

[69] Meyer, B. *Object-oriented software construction*, vol. 2. Prentice Hall New York, 1988.

[70] Moody, D. L. The physics of notations: a scientific approach to designing visual notations in software engineering. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (May 2010), vol. 2, ACM, pp. 485–486.

[71] Myers, D., and Salter, W. Industrial solutions find a place at CERN. *CERN Courier* (June 7, 2005).

[72] Naudet, Y., Latour, T., Guedria, W., and Chen, D. Towards a systemic formalisation of interoperability. *Computers in Industry vol. 61*, no. 2 (2010), pp. 176–185.

[73] Navigli, R., Velardi, P., and Missikoff, M. *Intelligent Technologies for Information Analyis.* Zhong, N. and Liu, J., 2013, ch. Web Ontology Learning and Engineering: An Integrated Approach, pp. 223–242.

[74] Neches, R., Fikes, R. E., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. R. Enabling technology for knowledge sharing. *AI magazine vol. 12*, no. 3 (1991), pp. 36–56.

[75] Nikiel, P. P., and Korcyl, K. Object mapping in the OPC-UA protocol for statically and dynamically typed programming languages. *Computing and Informatics* (2017). Submitted (copy on file with author).

[76] Noy, N. F., and McGuinness, D. L. Ontology development 101: A guide to creating your first ontology, March 2001. Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880, Stanford, CA.

[77] Object Management Group. OMG Systems Modeling Language (OMG SysML^{TM}), Version 1.4. `http://www.omg.org/spec/SysML/1.4/`, June 3, 2015.

[78] Oram, J., Goodman, D., and Sanders, G. Cost scaling of finely segmented filled aperture large telescope observatories: a tmt study. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2008), Conference 7017, Modeling, Systems Engineering, and Project Management for Astronomy III, p. 70170K.

[79] Ouaknine, J., and Worrell, J. Some recent results in metric temporal logic. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems* (2008), Springer-Verlag, pp. 1–13.

[80] Pessemier, W. Control systems for the Mercator telescope – Design and implementation of new control systems for the dome and the hydrostatic bearing. Master's thesis, Katholieke Hogeschool Sint-Lieven, Gent, 2008.

[81] Pessemier, W. Why semantics matter: or how to build smart machines using OPC UA. `http://opcconnect.opcfoundation.org/2015/12/why-semantics-matter/`, dec 2015. [Online; accessed 10 November 2016].

[82] Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H. Suitability assessment of OPC UA as the backbone of ground-based observatory control systems. In *Proceedings of the 13th International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)* (2011), pp. 1174–1177.

[83] Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H. Design and first commissioning results of PLC-based control systems for the Mercator telescope. Conference 8451, Software and Cyberinfrastructure for Astronomy II, International Society for Optics and Photonics (SPIE), p. 84512V.

[84] Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H. UAF: a generic OPC unified architecture framework. Conference 8451, Software and Cyberinfrastructure for Astronomy II, International Society for Optics and Photonics (SPIE), p. 84510P.

[85] Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H. Developing a PLC-friendly state machine model: lessons learned. Conference 9152, Software and Cyberinfrastructure for Astronomy III, International Society for Optics and Photonics (SPIE), p. 915208.

[86] Pessemier, W., Raskin, G., Prins, S., Saey, P., Merges, F., Padilla, J. P., Van Winckel, H., and Waelkens, C. Towards a new Mercator Observatory Control System. Conference 7740, Software and Cyberinfrastructure for Astronomy, International Society for Optics and Photonics (SPIE), p. 77403B.

[87] Pessemier, W., Raskin, G., Saey, P., Van Winckel, H., and Deconinck, G. Knowledge-based engineering of a PLC controlled telescope. Conference 9913, Software and Cyberinfrastructure for Astronomy IV, International Society for Optics and Photonics (SPIE), p. 991343.

[88] Pessemier, W., Raskin, G., Van Winckel, H., Deconinck, G., and Saey, P. A practical approach to ontology-enabled control systems for astronomical instrumentation. In *Proceedings of the 14th International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)* (2013).

[89] Pessemier, W., Raskin, G., Van Winckel, H., Saey, P., and Deconinck, G. Why semantics matter: a demonstration on knowledge-based control system design. In *Proceedings of the 15th International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)* (2015).

[90] Raskin, G., Bloemen, S., Morren, J., Padilla, J. P., Prins, S., Pessemier, W., Vandersteen, J., Merges, F., Østensen, R., and Van Winckel, H. MAIA, a three-channel imager for asteroseismology: instrument design. *Astronomy & Astrophysics vol. 559* (2013), p. A26.

[91] Raskin, G., Dubosson, R., Michaud, B., Pessemier, W., and Van Winckel, H. A new Nasmyth mirror mechanism increases the number of focal stations of the Mercator Telescope. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2012), Conference 8446, Ground-based and Airborne Instrumentation for Astronomy IV, International Society for Optics and Photonics (SPIE), pp. 46–54.

[92] Raskin, G., Van Winckel, H., Hensberge, H., Jorissen, A., Lehmann, H., Waelkens, C., Avila, G., De Cuyper, J.-P., Degroote, P., Dubosson, R., et al. HERMES: a high-resolution fibre-fed spectrograph for the Mercator telescope. *Astronomy & Astrophysics vol. 526* (2011), p. A69.

[93] Rowe, W. B. *Hydrostatic, Aerostatic, and Hybrid Bearing Design*. Elsevier, 2012.

[94] Rowley, J. E. The wisdom hierarchy: representations of the DIKW hierarchy. *Journal of information science* (2007).

[95] Russiniello, G.-B. *Système de commande et de contrôle pour télescopes de 1.2 mètre*. PhD thesis, Université de Genève, Genève, 1998.

[96] Schamai, W., Fritzson, P., and Paredis, C. J. Translation of UML state machines to Modelica: Handling semantic issues. *Simulation Journal* (2013).

[97] Schattkowsky, T., and Forster, A. On the Pitfalls of UML 2 Activity Modeling. In *Proceedings of the International Workshop on Modeling in Software Engineering* (Washington, DC, USA, 2007), MISE '07, IEEE Computer Society, pp. 8–14.

[98] Shan, L., and Zhu, H. Semantics of metamodels in UML. In *Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering, 2009 (TASE 2009)* (2009), IEEE, pp. 55–62.

[99] Silva, D. R., Angeli, G., Boyer, C., Sirota, M., and Trinh, T. Thirty Meter Telescope: observatory software requirements, architecture, and preliminary implementation strategies. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2008), Conference 7019, Advanced Software and Control for Astronomy II, International Society for Optics and Photonics (SPIE), p. 70190X.

[100] Stephenson, A. G., Mulville, D. R., Bauer, F. H., Dukeman, G. A., Norvig, P., LaPiana, L., Rutledge, P., Folta, D., and Sackheim, R. Mars climate orbiter mishap investigation board phase I report. *NASA, Washington, DC* (1999).

[101] Stepp, L., Daggert, L., and Gillett, P. Estimating the costs of extremely large telescopes. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2003), Conference 4840, Future Giant Telescopes, International Society for Optics and Photonics (SPIE), p. 309.

[102] Telles, M. *Python power! The comprehensive guide*. Cengage Learning, 2008.

[103] Theuerkorn, F. *Lightweight Enterprise Architectures*. CRC Press, 2004.

[104] TMT. Introduction to Thirty Meter Telescope (TMT) and TMT Observatory Software (OSW). Tech. rep., Thirty Meter Telescope, 2016. TMT.SFT.PRE.16.001.REL02.

[105] TMT. TMT Observatory Architecture Document. Tech. rep., Thirty Meter Telescope, 2016. TMT.SEN.DRD.05.002.CCR29.

[106] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Journal of Math vol. 58* (1936), pp. 345–363.

[107] Uschold, M., and Gruninger, M. Ontologies: Principles, methods and applications. *The knowledge engineering review vol. 11*, no. 2 (1996), pp. 93–136.

[108] Van de Laar, P., and Punter, T. *Views on evolvability of embedded systems*. Springer Science & Business Media, 2010.

[109] Van Deursen, A., Klint, P., and Visser, J. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices vol. 35*, no. 6 (2000), pp. 26–36.

[110] Vander Elst, L., and Pessemier, W. MERCATOR kan opnieuw geschiedenis schrijven. *ILYA – Magazine voor slimme oplossingen*, no. 21 (Jan-Feb 2016), pp. 7–8.

[111] Verzichelli, G. European Extremely Large Telescope (E-ELT) availability stochastic model: integrating FMEA, influence diagram, and Bayesian network together. In *Proceedings of SPIE Astronomical Telescopes and Instrumentation* (2016), Conference 9911, Modeling, Systems Engineering, and Project Management for Astronomy VI, International Society for Optics and Photonics (SPIE), p. 99110V.

[112] W3C. XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004. `https://www.w3.org/TR/xmlschema-2/`, 2004. [Online; accessed 21 October 2016].

[113] W3C. Simple part-whole relations in OWL Ontologies, W3C Editor's Draft 11 Aug 2005. http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/, 2005. [Online; accessed 1 November 2016].

[114] W3C. OWL 2 Web Ontology Language, Manchester Syntax (Second Edition), W3C Working Group Note 11 December 2012. http://www.w3.org/TR/owl2-manchester-syntax/, 2012. [Online; accessed 20 October 2016].

[115] W3C. OWL 2 Web Ontology Language Primer (Second Edition), W3C Recommendation 11 December 2012. https://www.w3.org/TR/owl2-primer/, 2012. [Online; accessed 5 October 2016].

[116] W3C. OWL 2 Web Ontology Language Profiles (Second Edition), W3C Recommendation 11 December 2012. https://www.w3.org/TR/owl2-profiles/, 2012. [Online; accessed 28 September 2016].

[117] W3C. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition), W3C Recommendation 11 December 2012. https://www.w3.org/TR/owl2-syntax/, 2012. [Online; accessed 28 September 2016].

[118] W3C. SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013. https://www.w3.org/TR/sparql11-query/, 2013. [Online; accessed 21 October 2016].

[119] W3C. RDF Schema 1.1, W3C Recommendation 25 Februrary 2014. https://www.w3.org/TR/rdf-schema/, 2014. [Online; accessed 28 September 2016].

[120] Wagner, D. A., Bennett, M. B., Karban, R., Rouquette, N., Jenkins, S., and Ingham, M. An ontology for State Analysis: Formalizing the mapping to SysML. In *2012 IEEE Aerospace Conference* (2012), IEEE, pp. 1–16.

[121] Wallace, P. The slalib library. In *Astronomical Data Analysis Software and Systems III* (1994), vol. 61, p. 481.

[122] Wikipedia. List of largest optical telescopes historically. http://en.wikipedia.org/wiki/List_of_largest_optical_telescopes_historically, 2016. [Online; accessed 15 September 2016].

Curriculum

I received my MSc degree in Engineering Technology (option Automation) from KU Leuven, Technology Campus Gent (former KAHO Sint-Lieven), Belgium in 2008. I started to work for the Institute of Astronomy of KU Leuven, after doing my Master's thesis on the Mercator Telescope there. In the first years, I developed a distributed component-based software framework to control the high-level functionality of the Mercator Telescope, and several components within this framework. When the lower-level functionality of the telescope needed to be refurbished, I started to investigate a more "industrial" approach by using modern soft-PLC's and model-based systems engineering tools, instead of the traditional approach of using Linux computers. Realizing the limitations of current practices, I conducted a PhD research project, the result of which is this thesis. My main interests are the automation and control of distributed real-time embedded systems, and systems engineering of complex systems in general. To contact me, send an e-mail to `wim.pessemier@outlook.com`.

List of publications

First author publications

Knowledge-based engineering of a PLC controlled telescope.
Pessemier, W., Raskin, G., Saey, P., Van Winckel, H., and Deconinck, G.
Proc. of *International Society for Optics and Photonics (SPIE)*, Vol. 9913, p. 991343, 2016.

Why semantics matter: or how to build smart machines using OPC UA.
Pessemier, W.
Online article for the OPC Foundation, December 2015.
`http://opcconnect.opcfoundation.org/2015/12/why-semantics-matter/`

Why semantics matter: a demonstration on knowledge-based control system design.
Pessemier, W., Raskin, G., Van Winckel, H., Saey, P., and Deconinck, G.
Proc. of *International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)*, 2015.

Developing a PLC-friendly state machine model: lessons learned.
Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H.
Proc. of *International Society for Optics and Photonics (SPIE)*, Vol. 9152, p. 915208, 2014.

A practical approach to ontology-enabled control systems for astronomical instrumentation.
Pessemier, W., Raskin, G., Van Winckel, H., Deconinck, G., and Saey, P.
Proc. of *International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)*, 2013.

Design and first commissioning results of PLC-based control systems for the Mercator telescope.
Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H.
Proc. of *International Society for Optics and Photonics (SPIE)*, Vol. 8451, p. 84512V, 2012.

UAF: a generic OPC unified architecture framework.
Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H.
Proc. of *International Society for Optics and Photonics (SPIE)*, Vol. 8451, p. 84510P, 2012.

Suitability assessment of OPC UA as the backbone of ground-based observatory control systems.
Pessemier, W., Deconinck, G., Raskin, G., Saey, P., and Van Winckel, H.
Proc. of *International Conference on Accelerators and Large Experimental Physics Control Systems (ICALEPCS)*, 2011.

Towards a new Mercator Observatory Control System.
Pessemier, W., Raskin, G., Prins, S., Saey, P., Merges, F., Padilla, J. P., Van Winckel, H., and Waelkens, C.
Proc. of *International Society for Optics and Photonics (SPIE)*, Vol. 7740, p. 77403B, 2010.

Control systems for the Mercator telescope – Design and implementation of new control systems for the dome and the hydrostatic bearing.
Pessemier, W.
Master's thesis, Katholieke Hogeschool Sint-Lieven, Gent, 2008.

Co-author publications

MAIA, a three-channel imager for asteroseismology: instrument design
Raskin, G., Bloemen, S., Morren, J., Padilla, J.P., Prins, S, Pessemier, W., ...
Journal of *Astronomy & Astrophysics*, Vol. 559, A26, 2013.

HERMES: a high-resolution fibre-fed spectrograph for the Mercator telescope
Raskin, G., Van Winckel, H., Hensberge, H., Jorissen, ..., Pessemier, W., ...
Journal of *Astronomy & Astrophysics*, Vol. 526, A69, 2011.

... and several more astronomy-oriented peer-reviewed papers.

Interviews

MERCATOR kan opnieuw geschiedenis schrijven
Cover article in ILYA, Ie-net Ingenieursmagazine, pp. 7-8, Feb. 2016.

Naar enige standaardisatie bij telescopen
Cover article in I-mag Ingenieursmagazine, Magazine of the Flemish Chamber of Engineers (VIK), pp. 5-8, Nov. 2011.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING (ESAT)
Research unit Electa (http://www.esat.kuleuven.be)
FACULTY OF SCIENCE
DEPARTMENT OF PHYSICS AND ASTRONOMY
Institute of Astronomy (http://www.ster.kuleuven.be)
Celestijnenlaan 200 D
B-3001 Leuven